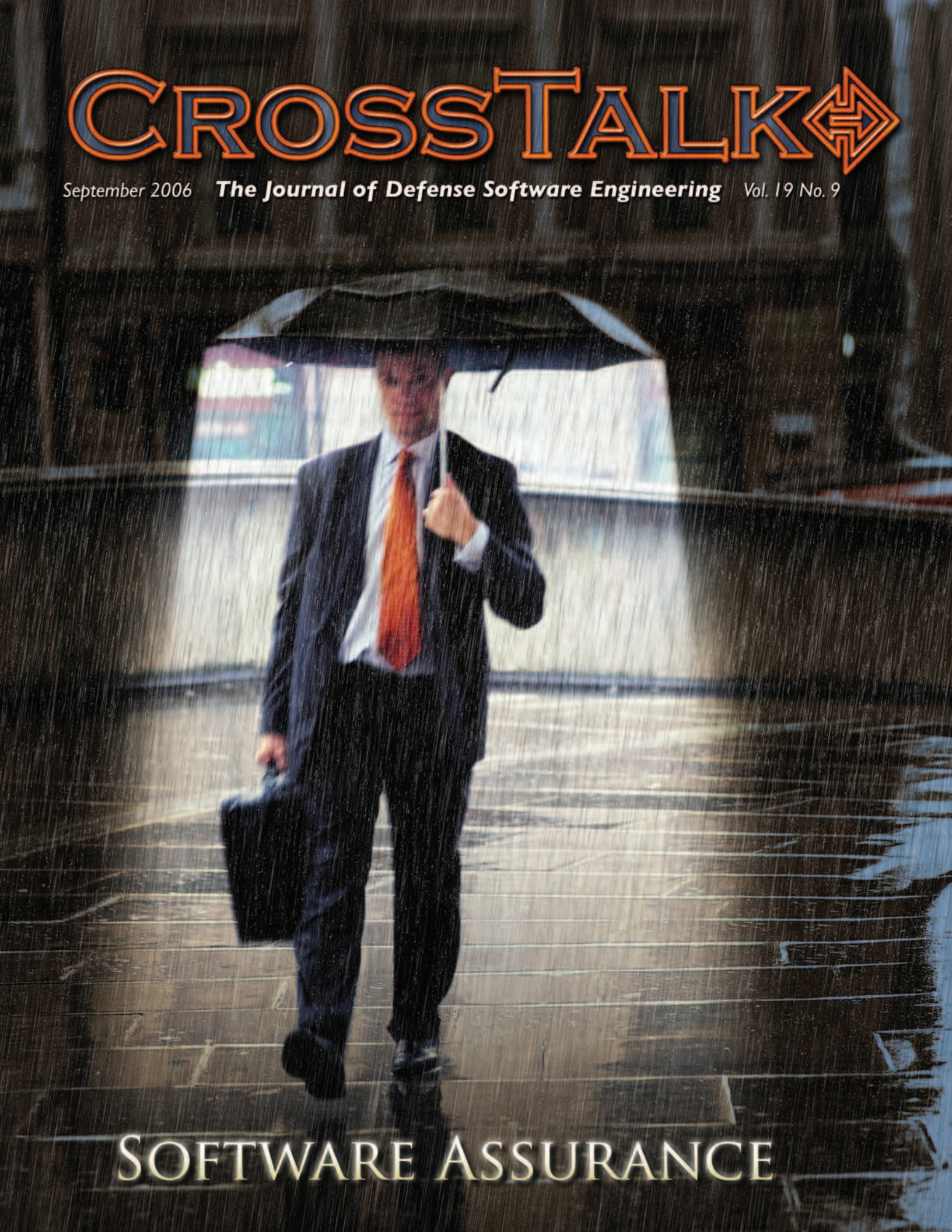


CrossTalk

September 2006 **The Journal of Defense Software Engineering** Vol. 19 No. 9

Software Assurance

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
SEP 2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk: The Journal of Defense Software Engineering. Volume 19, Number 9, September 2006	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**32**	

Software Assurance

ON THE COVER
Cover Design by Kent Bingham

Additional art services provided by Janna Jensen

Departments

CROSSTALK

76 SMXG Co-Sponsor	*Kevin Stamey*
309 SMXG Co-Sponsor	*Randy Hill*
402 SMXG Co-Sponsor	*Diane Suchan*
DHS Co-Sponsor	*Joe Jarzombek*
NAVAIR Co-Sponsor	*Jeff Schwalb*
PUBLISHER	*Brent Baxter*
ASSOCIATE PUBLISHER	*Elizabeth Starrett*
MANAGING EDITOR	*Kase Johnstun*
ASSOCIATE EDITOR	*Chelene Fortier-Lozancich*
ARTICLE COORDINATOR	*Nicole Kentta*
PHONE	(801) 775-5555
E-MAIL	crosstalk.staff@hill.af.mil
CROSSTALK ONLINE	www.stsc.hill.af.mil/crosstalk

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the U.S. Air Force (USAF), the U.S. Department of Homeland Security (DHS), and the U.S. Navy (USN). USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG), Ogden-ALC 309 SMXG, and Warner Robins-ALC 402 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection. USN co-sponsor: Naval Air Systems Command.

The USAF Software Technology Support Center (STSC) is the publisher of CROSSTALK, providing both editorial oversight and technical review of the journal. CROSSTALK's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 25.

517 SMXS/MDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

Software Assurance: Highlighting Changes Within Our Software Community of Practice

In an era riddled with asymmetric cyber attacks, claims about system reliability, integrity and safety must also include provisions for built-in security of the enabling software. To facilitate changes and adoption of requisite practices, the Departments of Homeland Security (DHS) and Defense (DoD) are investing in *Software Assurance* by partnering with software practitioners in industry, government, and academia to increase the availability and use of tools, knowledge, and guidance that will help improve the security and quality of the software used in national security systems and critical infrastructure.

How is Software Assurance influencing our software community of practice and addressing relevant needs? Over time, we learn to do what needs to change, and we share practices. From the users' perspective, we have learned that development and acquisition practices should only be categorized as *best practices* if they contribute to the delivery of safe and secure products, systems and services. Developers have continued questioning their assumptions about how software should be built. They have a growing understanding that functional correctness must be exhibited not only when the software executes under anticipated conditions but also when it is subjected to unanticipated, hostile conditions. Acquirers have a better understanding that more scrutiny is needed of their supply chains to reduce the risk exposures being passed to users of software and software-intensive systems. Interaction among practitioners continually refines and develops the elements of practice. This is why relevant knowledge and skills have limited shelf-lives that prompt competency *refresh* requirements.

An organization's software community of practice is critical to its success. The community may exist informally within and across business units and projects and often across organizational boundaries. To gain the most leverage, it maintains links outside the organization to strengthen its knowledge base. Communities of practice are organizational assets because of the knowledge they steward at their core and through the learning they inspire at their boundaries. The learning potential of an organization resides in the interaction of cores and boundaries in *constellations* or clusters of different communities of practices. Indeed, software assurance is derived from the application of integrated processes and practices from multiple disciplines, requiring software practitioners to interact with other communities of practice (such as systems engineering, program management, security, etc.).

Our software community of practice develops resources such as shared learning and practices. Several organizations facilitate the capture and transfer of knowledge critical to our software community practitioners. CROSSTALK functions as one of our software community's key conduits for transferring knowledge, and the DHS BuildSecurityIn Web site is evolving as an online resource at <http://BuildSecurityIn.us-cert.gov>.

The DHS Software Assurance Program provides a framework to shape a comprehensive strategy that addresses people, process, technology and acquisition throughout the software life cycle. Our efforts seek to shift the paradigm away from patch management and to achieve a broader ability to routinely develop and deploy trustworthy software products; contributing to the production of higher quality, more secure software. Through hosting and co-hosting various forums, we have leveraged collaborative efforts of public-private working groups. DHS initiatives, such as a software assurance common body of knowledge, guides for developers and acquisition managers, and the *BuildSecurityIn* Web site will continue to evolve and provide practical guidance on how to improve the quality, reliability, and security of software.

This DHS-sponsored issue of CROSSTALK addresses not only the value of software assurance, but also various methods in achieving it. I hope readers will take the time to understand and apply the principles and techniques. I encourage everyone to discover more about our Software Assurance efforts and learn more about proven security practices by reviewing our DHS BuildSecurityIn Web site and joining others in our expanding software assurance community of practice.

Joe Jarzombek, Project Management Professional (USAF Lt. Col., Retired)
Director for Software Assurance
National Cyber Security Division
Department of Homeland Security

Security in the Software Life Cycle

Joe Jarzombek
Department of Homeland Security

Karen Mercedes Goertzel
Booz Allen Hamilton

As a freely downloadable reference document, "Security in the Software Life Cycle: Making Application Development Processes – and Software Produced by Them – More Secure" presents key issues in the security of software and its development processes. It introduces a number of process improvement models, risk management and development methodologies, and sound practices and supporting tools that have been reported to help reduce the vulnerabilities and exploitable defects in software and diminish the possibility that malicious logic and trap doors may be surreptitiously introduced during its development. No single practice, process, or methodology offers the universal silver bullet for software security. "Security in the Software Life Cycle" has been compiled as a reference document with practical guidance intended to tie it together and inform software practitioners of a number of practices and methodologies from which they can evaluate and selectively adopt to reshape their development processes to increase not only the security but also the quality and reliability of their software applications, services, and systems, both in development and in deployment.

In an era riddled with asymmetric cyber attacks, claims about system reliability, integrity and safety must also include provisions for built-in security of the enabling software. The Department of Homeland Security (DHS) Software Assurance Program has undertaken to partner with software practitioners in industry, government, and academia to increase the availability and use of tools, knowledge, and guidance that will help improve the security and quality of the software they produce. In addition to its BuildSecurityIn Web portal [1] and Software Assurance Common Body of Knowledge [2], the DHS Software Assurance Program is publishing *Security in the Software Life Cycle: Making Application Development Processes – and Software Produced by Them – More Secure* [3] (freely downloadable from the DHS BuildSecurityIn portal).

Background

Software is ubiquitous. Many functions in the public and private sectors depend on software to perform correctly even during times of crisis despite attempts to subvert or compromise its functions. Software is relied on to handle sensitive and high-value data on which users' privacy, livelihoods, and lives depend. Our national security and homeland security increasingly depend on ever more complex, interconnected, software-intensive information systems that use Internet-exposed networks as their common data bus. Software enables and controls systems that run the nation's critical infrastructure: electrical power grids; water treatment and distribution systems; air traffic control and transportation signaling systems; nuclear, biological, and chemical laboratories and manufacturing plants; medical and emergency response systems; financial systems; and other critical functions, such as law enforcement, criminal justice, immigration, and, increasingly, voting. Software also protects other software. The majority of filtering routers, firewalls, encryption systems, and intrusion detection systems are implemented, at least in large part, through the use of software.

> *"The key to secure software is the development process used to conceive, implement, deploy, and update/sustain it."*

Perhaps because of this dependence, nation-state adversaries, terrorists, and criminals have joined malicious and recreational attackers in targeting this growing multiplicity of software-intensive systems. These new threats are both better resourced and highly motivated to discover and exploit vulnerabilities in software. The National Institute of Standards and Technology's (NIST) special publication 800-42, *Guideline on Network Security Testing* [4] sums up the challenge: Many successful attacks exploit errors (bugs) in the software code used on computers and networks. This is why software security matters. As more and more critical functions become increasingly dependent on software, that software becomes an extremely high-value target.

The objective of *Security in the Software Life Cycle* is to inform developers about existing processes, methods, and techniques that can help them to specify, design, implement, configure, update, and sustain software that is able to accomplish the following:

1. Resist or withstand many anticipated attacks.
2. Recover rapidly and mitigate damage from attacks that cannot be resisted or withstood.

The key to secure software is the development process used to conceive, implement, deploy, and update/sustain it. A *security-enhanced* software development life cycle process includes practices that not only help developers root out and remove exploitable defects (e.g., vulnerabilities) in the short term, but also, over time, increase the likelihood that such defects will not be introduced in the first place.

Security in the Software Life Cycle also addresses the risks associated with the software supply chain, including risks associated with selection and use of commercial off-the-shelf and open source software components, software pedigree (or more accurately the inability to determine software pedigree), and outsourcing of software development and support to offshore organizations and unvetted domestic suppliers.

Many security defects in software could be avoided if developers were better equipped to recognize the security implications of their design and implementation choices. Quality initiatives that reduce the number of overall defects in software are a good start, as many of those defects are security-related vulnerabilities. However, quality-based methods that focus only on improving the *correctness* of software seldom result in secure software. This is because in quality terms, *correctness* means correctness of the software's operation under anticipated, *normal* conditions.

Attacks on software and its environment typically create *un*anticipated, *ab*normal conditions. Moreover, rather than targeting single defects, attackers often leverage combinations of what seem to be correct functionalities and interactions in software. Individually, none of those functionalities or interactions might be problematical. Only when they are combined in an unexpected or unintended way does an exploitable vulnerability manifest. To a great extent, software security relies on the developer's ability to anticipate the unexpected.

A software security program must include training and educating developers to do exactly that, and to recognize the security implications not only of simple defects, but also of *abnormal* series and combinations of functions and interactions within their software, and between their software and other entities. Developers need to work within a development process framework that not only allows but also encourages them to analyze their software and recognize the security vulnerabilities that manifest from seemingly *minor* individual defects and coding errors, as well as their larger design and implementation choices.

Security Enhancement of the Software Development Life Cycle

Security enhancement of development life cycle processes and practices requires a shift in emphasis and an expansion of scope of existing development activities so that security is given equal importance as other desirable properties, such as usability, interoperability, reliability, safety, and performance. These shifts in emphasis and scope will affect the life cycle in the following ways:

1. **Requirements.** The review of the software's functional requirements will include a security vulnerability and risk assessment that forms the basis for capturing the set of general non-functional security requirements (often stated in terms of constraints on functionality) or *derived* security provisions (not explicitly specified in customer documents) that will mitigate the identified risks. These high-level, non-functional/constraint requirements will then be *translated* into more specific requirements for functionalities and functional parameters that otherwise may not have been included in the software specification.
2. **Design and implementation.** Design assumptions and choices that determine how the software will operate and how different modules/components will interact should be analyzed and adjusted to minimize the exposure of those functions and interfaces to attackers. Implementation choices will start with selection of only those languages (or language constructs), libraries, tools, and reusable components that are determined to be free of vulnerabilities, or for which an effective vulnerability mitigation can be realistically implemented. Bugs in source code must be flagged not just when they result in incorrect functionality, but when otherwise correct functionality is able to be corrupted by unintended and unusual inputs or configuration parameters.

> *"A key component of risk-driven software development occurs at the beginning of the development process: threat modeling."*

3. **Reviewing, evaluating, and testing.** Security criteria will be generated for every specification, design, and code review, as part of the software/system engineering evaluation and selection of acquired and reused components, and as part of the software's unit and integration test plans. Life cycle phase-appropriate security reviews and/or tests will establish whether all software artifacts (e.g., code, documentation) have satisfied their security exit criteria at the end of one life cycle phase before development is allowed to move into the next phase.
4. **Distribution, deployment, and support.** Distribution will be preceded by careful code clean-up to remove any residual security issues (e.g., debug hooks, hard-coded credentials). Preparation for distribution will also include documenting the secure configuration parameters that should be set when the software is installed; these include both parameters for the software itself and for any components of its execution environment (e.g., file system and Web server access controls, application firewall) that it will rely upon to protect it in deployment. *Secure by default* distribution configurations and trustworthy distribution mechanisms will be adopted. Ongoing, post-deployment, vulnerability and threat assessments, and forensic analyses of successful attacks will be performed for the software and its environment. New requirements to be satisfied in future releases will be formulated based on the assessment and analysis findings.

Risk-Driven Requirements Engineering

It is easier to produce software that can resist and recover from attacks when risk management activities and checkpoints are integrated throughout the software life cycle. A key component of risk-driven software development occurs at the beginning of the development process: threat modeling. To identify and prioritize mitigation steps that must be taken, developers first need to recognize and understand the threats to their software, including the threat agents, anticipated attack patterns, vulnerabilities likely to be exploited, and assets likely to be targeted, and they need to assess the level of risk that the threat will occur and its potential impact if it does. Several freely downloadable methodologies have emerged to support the developer in modeling threats to applications and other software-intensive systems, including the following:

- Microsoft's ACE (Application Consulting and Engineering) Threat Analysis and Modeling (also called Threat Modeling Version 2.0) [5].
- European Union Consultative Objective Risk Analysis System (CORAS) [6, 7, 8] and Research Council of Norway Model-Driven Development and Analysis of Secure Information Systems (SECURIS) [9, 10].
- Practical Threat Analysis (Technologies' Calculative Threat Modeling Methodology) [11].

- Trike: A Conceptual Framework for Threat Modeling [12].
- National Aeronautics and Space Administration Software Security Assessment Instrument [13, 14].
- Visa USA Payment Application Best Practices [15].

A number of other system-level methodologies and tools are also in use for risk analysis of software-intensive systems, including NIST's Automated Security Self Evaluation Tool, Carnegie Mellon University's Operationally Critical Threat Asset and Vulnerability Evaluation-Secure, and Siemens/Insight Consulting Central Computer and Telecommunications Agency Risk Analysis and Management Method. These methodologies are also introduced in *Security in the Software Life Cycle*.

Security-Enhanced Process Models and Development Methodologies

The use of repeatable process improvement models has been demonstrated to improve the efficiency and adaptability of software development life cycle activities and the overall quality of software by reducing the number and magnitude of errors. The Software Engineering Institute's Capability Maturity Model® (CMM®) and General Electric's Six Sigma are the most widely used process improvement models. A number of CMM variants have been tailored for specific communities or problem spaces, e.g., CMM Integration (CMMI®), integrated-CMM (iCMM), and System Security Engineering (SSE)-CMM, which is also an international standard [16].

Beyond the SSE-CMM, a number of efforts have been undertaken to adapt or extend existing maturity models or define new process improvement models that have security-enhancement as their main objective. These include the following:

- Safety and Security Extensions to CMMI/iCMM [17].
- Revised International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) Standard 15026 *System and Software Assurance*, which adds security assurance activities to ISO/IEC 15288 system life cycle and ISO/IEC 12207 software life cycle processes.
- Microsoft Security Development Lifecycle (SDL) [18, 19].
- Comprehensive, Lightweight Application Security Process (CLASP) [20].
- Carnegie Mellon University Software Engineering institute (SEI) Secure Team Software Process℠ (TSP℠ Secure) [21].

The CMM and ISO/IEC process models are defined at a higher level of abstraction than SDL and CLASP, which bridge the abstraction gap between CMM-level models and software development methodologies. It is quite possible to implement both a security-enhanced CMMI or iCMM as an overarching framework in which SDL or CLASP processes can be executed. SDL or CLASP could then itself provide a framework in which life cycle, phase-specific, security-enhanced methods such as Model-Driven Architecture (MDA) or Aspect-Oriented Programming (AOP) could be employed. Indeed, these higher-level models are intended to be *methodology-neutral* and to accommodate development using any of a variety of methodologies.

Using Familiar Development Methodologies in Ways that Improve Software Security

Unlike process improvement models, software development methodologies are specific in purpose and applicability. Efforts have been made to use existing methodologies in ways that are expressly intended to support the engineering of security functionality in software. In a few cases, efforts have been made to adapt these methodologies to expressly improve the security of the software produced.

MDA

Defined by the Object Management Group (OMG), MDA automatically transforms Unified Modeling Language (UML) models into platform-specific models and generates a significant portion of the application source code (the eventual goal is to generate whole applications). By using MDA, developers need to write less code, and the code they do write can be less complex. As a result, software contains fewer design and coding errors (including errors with security implications). Researchers outside OMG are looking at ways to add security to MDA by combining it with elements of Aspect-Oriented Modeling or by defining new UML-based security modeling languages to be used in producing MDA-based secure design models. Both Interactive Objects' ArcStyler [22] and IBM/Rational's Software Architect [23] support MDA-based security modeling, model checking, and automatic code generation from security models.

Object-Oriented Modeling With UML

Unlike security functions, security properties in object-oriented modeling are treated as generic nonfunctional requirements and thus do not require specific security artifacts. UML, which has become the *de facto* standard language for object-oriented modeling, lacks explicit syntax for modeling the misuse and abuse cases that can help developers predict the behavior of software in response to attacks. Recognizing these omissions, some UML profiles have been published that add expressions for modeling security functions, properties, threats and countermeasures, etc., in UML. SecureUML [24] provides extensions to support modeling of access controls and authorization. UMLSec [25] adds both access control/authorization modeling extensions and support for vulnerability assessment of UML models. The CORAS UML profile for security assessment [26], which has been adopted as a recommended standard by the OMG, is the most directly applicable to software security needs as it provides UML extensions for modeling threats and countermeasures.

Aspect-Oriented Software Development (AOSD)

Object-oriented development requires security properties and functions to be associated with each individual object in which that property/function must be exhibited. Security properties such as *nonsubvertability* and functions such as code signature validation are *crosscutting*, i.e., they are required in multiple objects. In object-oriented development, the developer would have to replicate and propagate the expressions of such cross-cutting security properties and functions to every object to which they pertain. This represents an unnecessarily high level of effort, both to initially specify and even more to make changes to cross-cutting security functions and properties, because such changes would also need to be replicated, propagated, and tracked.

AOM and Design extend the expressions possible in object-oriented modeling so that cross-cutting properties and functions are able to be expressed only once in a single modular component of the software model and design specification. This cross-cutting component is then referred to by all the components/

® Capability Maturity Model, CMM, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
℠ Team Software Process and TSP are service marks of Carnegie Mellon University.

objects to which the property/ function pertains. AOP allows the developer to write and maintain the cross-cutting component at a single location in the code. The AOP tools then automatically replicate and propagate that cross-cutting code to all appropriate locations throughout the code base. This automated approach reduces the potential that the developer may inadvertently omit the cross-cutting code from some components/objects.

Agile Methods and Secure Software

Can Agile development produce secure software? As with structured development methodologies, Agile methods seek to promote development of high-quality software. With the exception of Lean Development, the collection of methodologies that falls under the umbrella of *Agile Methods* all share a commitment to the core principles of the Agile Manifesto [27]. According to the Agile Manifesto, *Agility is enhanced by continuous attention to technical excellence and good design*. Proponents often cite key practices in many agile methods that help reduce the number of exploitable defects that are introduced into software practices such as enforced coding standards, simple designs, pair programming, continuous integration, and test-driven development (also known as *continuous testing*).

It has been suggested that Agile methods can support risk-driven software engineering *if* – and this a very big *if* – the requirements-based, functional test-driven development approach that underpins all Agile methods is extended to include the following:

- Redefine the customer (who drives all response to change) to include the stakeholders such as risk managers, certifiers, and accreditors responsible for enforcing security policy and preventing and responding to attacks on software.
- Expand agile testing to accommodate both requirements-based and risk-based tests (see Software Security Testing).

Security in the Software Life Cycle includes an extensive discussion of whether, and, if so, how and when Agile methods can be used for secure software development without violating the Agile Manifesto's core principles.

Formal Methods and Secure Software

Formal methods use mathematical proofs and precise specification and verification methods to clarify the expression and understanding of software requirements and design specifications. The strict mathematical foundations of formal methods create a basis for positive assurance that the software's specifications are consistent with its formally modeled properties and attributes.

Formal methods have been used successfully to specify and prove the correctness and internal consistency of security function specifications (e.g., authentication, secure input/output, mandatory access control) and security-related trace properties (e.g., secrecy). However, to date, none of the widely used formal languages or techniques are explicitly devoted to specification and verification of *non*-trace security properties, such as non-subvertability or correct, predictable behavior in the face of unanticipated changes in environment state.

> *"Security analysis and testing are most effective when a multifaceted approach is used that employs as wide a variety of techniques and technologies as time and resources allow."*

Software Security Testing

Software security testing verifies that the software produced is indeed secure. It does this by observing the way in which software systems and the components they contain behave in isolation and as they interact. The main objectives of software security testing are: 1) Detection of security defects, coding errors, and other vulnerabilities including those that manifest from complex relationships among functions, and those that exist in obscure areas of code, such as dormant functions; 2) demonstration of continued secure behavior when subjected to attack patterns; and 3) verification that the software consistently exhibits its required security properties and functional constraints under both normal and hostile conditions. However, a security requirements testing approach alone demonstrates only whether the stated security requirements have been satisfied, regardless of whether those requirements were adequate. Most software specifications do not include negative and constraint requirements such as *no failure may result in the software dumping core memory or the software must not write input to a buffer that is larger than the memory allocated for that buffer*.

Risk-based testing takes into account the fact that during the time between requirements capture and integration testing, the threats to which the software will be subject are likely to have changed. For this reason, risk-based testing includes subjecting the software to attack patterns that are likely to exist at time of deployment, not just those that were likely at time of requirements capture. In practical terms, many of the same tools and techniques used for functional testing will be useful during security testing. The key difference will be the test scenarios exercised in risk-based tests. The software's security test plan should include test cases (including cases based on abuse and misuse cases) that exercise areas and behaviors that may not be exercised by functional, requirements-based testing. These security test cases should attempt to demonstrate the following:

- The software behaves consistently and securely under all conditions, both expected and unexpected.
- If the software fails, the failure does not leave the software, its data, or its resources exposed to attack.
- Obscure areas of code and dormant functions cannot be exploited or compromised.
- Interfaces and interactions among components at the application, framework/middleware, and operating-system levels are consistently secure.
- Exception and error handling resolve all faults and errors in ways that do not leave the software, its resources, its data, or its environment vulnerable to unauthorized modification (disclosure) or denial of service.

Security analysis and testing are most effective when a multifaceted approach is used that employs as wide a variety of techniques and technologies as time and resources allow. These techniques may include the following:

- **White box security reviews and tests.** Performed on source code, white box testing techniques include code security review (direct code analysis, property-based testing); source code fault injection with fault propagation analysis; and automated compile-time detection.
- **Black box security test techniques.**

Targeting individual binary components and/or the software system as a whole, black box techniques are the only testing option when source code is not available. Black box techniques include software penetration testing, security fault injecting of binaries, fuzz testing, and automated vulnerability scanning.

- **Reverse engineering.** Disassembly and decompilation generate, in the former case, assembler code, and in the latter, source code – both of which can then be analyzed for security-relevant implementation errors and vulnerabilities. Decompiled source code can be subjected to standard white box security tests and tools. Reverse-engineering is often more difficult and time-consuming than other software security test techniques; many commercial software products use obfuscation techniques to deter reverse-engineering. Like formal methods, the level of effort required makes reverse-engineering practical only for the examination of high-consequence, high-confidence, or high-risk components. Even then, there is no 100 percent guarantee of success.

The DHS-funded NIST Software Assurance Metrics and Tool Evaluation program [28] has developed a taxonomy of security testing tool categories that include a database of profiles of commercial and open source tools within each category. This database is being used to capture the results of tool evaluations and measurements of tool effectiveness, and it is used to conduct a gap analyses of tool capabilities and methods.

Both DHS and the DoD continue to dedicate resources toward achieving software assurance [29], and individual organizations can implement practices today to contribute toward software assurance in the near future.

Sound Practices for Security Enhancing Life Cycle Activities

Practitioners and program managers need to first understand *what secure software is* [30]. Appendix G of *Security in the Software Life Cycle* presents a collection of security principles and sound practices that have been expounded on by respected practitioners of secure software development in the private, public, and academic sectors, in the U.S. and abroad. These principles and practices enable the insertion of security considerations into all phases of the software life cycle. Appendix G also describes practices that span life cycle phases such as secure configuration management, security-minded quality assurance, security training and education of developers, and selection and secure use of frameworks, platforms, development tools, libraries, and languages. Developers who start applying these practices today should start to see improvements in their software's security, as well as its quality. This is true even if the organizations they work for never commit to adopting a security-enhanced, structured development methodology or process improvement model.

Summary

With its increasing exposure and criticality, software has become a high-value target not just for malicious and recreational hackers, but for highly motivated, well-resourced cyber-terrorists, cyber-criminals, and information warfare adversaries. At the same time, user expectations (real or perceived) that new functionality can be delivered near-instantaneously has driven software suppliers to adopt ever-shortening release schedules and agile development methods that do not allow sufficient time for careful specification, design, coding, and testing. As a result, the software they produce is inordinately convoluted and complex, with seemingly infinite possible internal states and a multiplicity of flaws and defects. All of these factors make software increasingly vulnerable to the intensifying threats that surround it.

"... software has become a high-value target not just for malicious and recreational hackers, but for highly motivated, well-resourced cyber-terrorists, cyber-criminals, and information warfare adversaries."

Developers need to start questioning their assumptions about how software should be built. They need to understand that functional correctness must be exhibited not only when the software executes under anticipated conditions, but also when it is subjected to *un*anticipated, *hostile* conditions. *Security in the Software Life Cycle* provides developers with information that can help them achieve a two-phase security enhancement of their software processes. For the first phase, Appendix G describes sound practices and principles that developers can begin to apply immediately throughout the life cycle. These practices and principles are intended to *raise the floor*, enabling developers to achieve a basic level of security in their software processes. The security-enhanced process improvement models and life cycle methodologies in the rest of the document are intended to help developers *raise the ceiling* over the longer term by making additional, significant increases in the security of their processes and by adding structure and repeatability to further security-enhancement of those processes.◆

References

1. United States. Dept. of Homeland Security. BuildSecurityIn Portal. National Cyber Security Division <https://buildsecurityin.us-cert.gov/>.
2 Redwine, Jr., Samuel T. ed. Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software (DRAFT Version 1.1). Washington, D.C.: Department of Homeland Security, July 2006 <https://buildsecurityin.us-cert.gov/>.
3 Goertzel, K.M., et al. Security in the Software Life Cycle: Making Application Development Processes – and Software Produced by Them – More Secure (DRAFT Version 1.1). Washington, D.C.: Department of Homeland Security, July 2006 <https://buildsecurityin.us-cert.gov/>.
4. National Institute of Standards and Technology. Guideline on Network Security Testing. Special Publication 800-42. Oct. 2003 (intended solely as a source of information and guidance, not as a proposed standard, directive, or policy from DHS; descriptions of processes, methodologies, and technologies containing this document should not be interpreted as formal endorsements by DHS) <http://csrc.nist.gov/publications/nistpubs/800-42/NIST-SP800-42.pdf>.
5. Microsoft. Microsoft Threat Analysis & Modeling. v2.0 BETA2 <www.microsoft.com/downloads/details.asp

x?FamilyID=570dccd9-596a-44bc-bed7-1f6f0ad79e3d&DisplayLang=en>.
6. CORAS. A Platform for Risk Analysis of Security Critical Systems <www2.nr.no/coras/>.
7. CORAS. The CORAS Project <http://coras.sourceforge.net/>.
8. CORAS. A Tool-Supported Methodology for Model-Based Risk Analysis of Security Critical Systems <http://heim.ifi.uio.no/~ketils/coras/>.
9. SECURIS. Model-Driven Development and Analysis of Secure Information Systems <www.sintef.no/content/page1_1824.aspx>.
10. The SECURIS Project. Model-Driven Development and Analysis of Secure Information Systems <http://heim.ifi.uio.no/~ketils/securis/index.htm>.
11. PTA Technologies. Practical Threat Analysis for Securing Computerized Systems <www.ptatechnologies.com>.
12. Trike. A Conceptual Framework for Threat Modeling <http://dymaxion.org/trike/> and <www.octotrike.org/>.
13. National Aeronautics and Space Administration. Reducing Software Security Risk <http://rssr.jpl.nasa.gov>.
14. "Reducing Software Security Risk through an Integrated Approach." University of California-Davis <http://seclab.cs.ucdavis.edu/projects/testing/>.
15. "Visa USA Cardholder Information Security Program: Payment Applications." Visa <http://usa.visa.com/business/accepting_visa/ops_risk_management/cisp_payment_applications.html>.
16. International Standard ISO/IEC 21827, System Security Engineering (SSE)-CMM® <http://www.issea.org> and <http://www.SSE-CMM.org>.
17. Federal Aviation Administration Integrated Process Group. Safety and Security Extensions to Integrated Capability Maturity Models. Washington D.C.: Sept. 2004 <http://faa.gov/ipg/news/docs/SafetyandSecurity Ext-FINAL.pdf>.
18. Lipner, Steve, and Michael Howard. "The Trustworthy Computing Security Development Lifecycle." Microsoft <http://msdn.microsoft.com/security/sdl>.
19. Howard, Michael. "How Do They Do It?: A Look Inside the Security Development Lifecycle at Microsoft." MSDN Magazine: The Microsoft Journal for Developers. 20.11 (Nov. 2005) <http://msdn.microsoft.com/msdnmag/issues/05/11/SDL/default.aspx>.
20. "Comprehensive, Lightweight Application Security Process." Secure Software <https://securesoftware.custhelp.com/cgi-bin/securesoftware.cfg/php/enduser/doc_serve.php?2=CLASP>.
21. SEI Software Process (TSP) for Secure Systems Development <www.sei.cmu.edu/espltsp-secure.presentation/>.
22. "ArcStyler Overview." Interactive Objects <www.interactive-objects.com/products/arcstyler-overview>.
23. "Rational Software Architect." IBM <www.306.ibm.com/software/awdtools/architect/swarchitect/>.
24. Lodderstedt, Torsten. "Model Driven Security from UML Models to Access Control Architectures." Diss. Albert-Ludwigs-Universität, 2003 <http://deposit.ddb.de/cgi-bin/dokserv?idn=971069778&dok_var=d1&dok_ext=pdf&filename=971069778.pdf>.
25. UMLsec <http://www4.in.tum.de/~umlsec/>.
26. "The CORAS UML Profile." CORAS <http://coras.source-forge.net/uml_profile.html>.
27. Agile Alliance. "Manifesto for Agile Software Development." Agilemanifesto <http://agilemanifesto.org/>.
28. National Institute of Standards & Technology. "Software Assurance Metrics and Tool Evaluation." <http://samate.nist.gov/ index.php>.
29. "Software Assurance." Wikipedia <http://en.wikipedia.org/wiki/software-assurance>.
30. Goertzel, K.M. "What Is Secure Software?" IA Newsletter (Summer 2006) <http://iac.dtic.mil/iatac>.

About the Authors

Joe Jarzombek is the Director for Software Assurance in the Department of Homeland Security (DHS) National Cyber Security Division. He leads government interagency efforts with industry, academia, and standards organizations to shift the security paradigm away from patch management by addressing security needs in work force education and training, research and development (especially diagnostic tools), and development and acquisition practices. After retiring from the U.S. Air Force as a Lt. Col. in program management, Jarzombek worked in the cyber security industry as vice president for product and process engineering. He later served in two software-related positions within the Office of the Secretary of Defense prior to accepting his current DHS position. As a Project Management Professional, Jarzombek has spoken extensively on measurement, software assurance, and acquisition topics. He encourages further review of DHS-sponsored software assurance efforts via the BuildSecurityIn Web site.

National Cyber Security Division
Department of
Homeland Security
Phone: (703) 235-5126
Fax: (703) 235-5962
E-mail: joe.jarzombek@dhs.gov

Karen Mercedes Goertzel is a software security subject-matter expert supporting the director of the Department of Homeland Security's Software Assurance Program, and has provided support to the Department of Defense's Software Assurance Tiger Team. She was project manager for the Defense Information Systems Agency Application Security Support Task, and currently leads the team developing the National Institute of Standards and Technology's special publication 800-95, *Guide to Secure Web Services*. In addition to software assurance and application security, Goertzel has extensive experience in trusted systems and cross-domain information sharing solutions and architectures, information assurance (IA) architecture and cyber security architecture, risk management, and mission assurance. She has written and spoken extensively on security topics and IA topics both in the United States and abroad.

Booz Allen Hamilton
8283 Greensboro DR
H5061
McLean, VA 22102
Phone: (703) 902-6981
Fax: (703) 902-3537
E-mail: goertzel_karen@bah.com

When Computers Fly, It Has to Be Right: Using SPARK for Flight Control of Small Unmanned Aerial Vehicles

Lt. Col. Ricky E. Sward, Ph.D., Lt. Col. Mark Gerken, Ph.D., and 2nd Lt. Dan Casey
U.S. Air Force Academy

One approach to software assurance is to use an annotated language such as SPARK. For safety critical software programs such as Unmanned Aerial Vehicle flight control software, the risk of software failure demands high assurance that the software will perform its intended function. Using an example from work being done at the U.S. Air Force Academy, this article describes SPARK and the formal process of proving correctness of software implementations.

In recent years, we have seen small Unmanned Aerial Vehicles (UAVs) emerge onto the battlespace. These small UAVs, which weigh less than 50 pounds, carry five to 10 pounds of payload, have mission endurances of about one hour, and field missions to support tactical surveillance and reconnaissance operations. As the number of missions for these small UAVs increases, more reliance will be placed on the computer systems that will control these aircraft. It is already possible for a computer program to create a flight plan for a UAV and place the craft in an orbit to observe an item of interest. Flight control computer programs are *safety critical* because errors in these programs could result in the loss of the UAV or cause damage to buildings and/or people on the ground [1].

Because of the critical nature of this software, development processes that result in *high-integrity* software (software systems that have a very low probability of failure) [1] must be used. Verification and validation (V&V) is often an important part of these processes. However, V&V of a safety-critical computer program can consume up to 50 percent of the development time [2]. One approach to reuse dedicated V&V schedule time while still assuring software quality is to use an annotated programming language. For example, the SPARK programming language and its associated tools increase the quality of software developed while reducing overall development time [3, 4]. This approach to software development can easily become a valuable tool when dealing with safety-critical systems. The SPARK language, which is a commercial product available from Praxis High Integrity Systems[1], is ideal for systems such as UAV flight planning software. The SPARK language has been used on many successful software development projects such as the C-130J [4] and is the language we have used on our UAV project. SPARK is an annotated language similar to the annotated Ada language [5] and the Larch annotated language for C [6]. The annotations in SPARK create extra work for the development team, but it has been shown the return on time investment can be as high as an 80 percent reduction in testing costs [4].

This article briefly discusses the SPARK programming language and development process. We examine a safety critical software example developed for small UAVs developed in a senior-level computer science course at the Air Force Academy. We elucidate the SPARK process by examining a small section of the UAV control software dedicated to orbital control.

Background

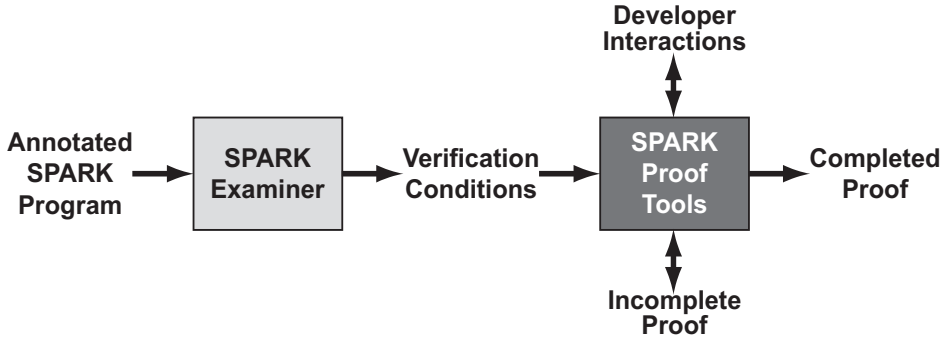
SPARK is an annotated subset of the Ada programming language, and every SPARK program can be compiled by an Ada compiler. However, SPARK includes several restrictions and rules governing the use of various programming constructs. These rules not only serve to simplify the V&V process, but also have a secondary benefit of helping assure good coding practice. For example, SPARK does not support dynamic allocation of memory so things such as pointers and heap variables are not supported. The use of GOTO statements is also not supported and there are restrictions on EXIT statements in loops. These rules and restrictions in SPARK allow the developer to predict the exact outcome when the code is executed. This prediction ability is a key feature of SPARK.

In order to show that the code developed has a very low error rate, SPARK uses formal, mathematical techniques to prove that the code is correct. The proof of correctness relies on the mathematical description of what is true before the code is executed and what is true after the code is executed. These are referred to as preconditions and postconditions, respectively. SPARK includes annotations that allow the programmer to embed the *precondition* and *postcondition* into a segment of code in the form of comments.

The proof of correctness uses a technique that begins with the postcondition and works backward through a segment of code, *hoisting* the postcondition up through the SPARK code [1]. Since the code is predictable, it is possible to determine a general effect of each statement and to reverse that effect, working backward from the postcondition toward the precondition. The result of this hoisting is called a *verification condition* in SPARK. If it can be shown that the precondition for the segment of code implies the verification condition developed from the hoisting process, the proof of correctness is complete.

SPARK includes several tools that help with the proof of correctness. The Examiner tool performs the hoisting operation and produces a collection of

Figure 1: *SPARK Tool Set*



verification conditions. The development team uses the Simplifier tool to reduce the verification conditions as much as possible before attempting to prove that they are implied by the precondition. The Proof Checker tool is used to prove that the verification conditions imply the precondition for a segment of code.

Figure 1 shows the process used by a software developer. The developer is responsible for annotating the code with preconditions and postconditions. The developer then executes the Examiner tool on the annotated SPARK code, and the Examiner returns the verification conditions. The developer then executes the Simplifier tool on the verification conditions, and the Simplifier returns the simplified verification conditions. The developer then executes the Proof Checker on the verification conditions to see if the tool can build a proof of program correctness automatically. If such a proof cannot be built, then the developer attempts to build a proof manually. In this situation, either the code is incorrect or the approach to constructing such a proof is insufficient. An incomplete proof does not necessarily mean that the code is incorrect, but a complete proof does mean the code is correctly implemented.

The burden placed on the developer is to build correct preconditions and postconditions, as well as correct code. If a proof cannot be built automatically for the code, then the developer will also need to examine the code or the verification conditions to see if a proof can be built manually. This may seem like an undue burden on the developer, but the return on this investment can be as much as an 80 percent reduction in costs during the testing phase [4] due to the fact that the code being developed is provably correct while being built.

It should be noted that since the preconditions and postconditions are written by the code developer, they are subject to human error. Therefore, these preconditions and postconditions should be reviewed and verified by a separate software development team. This moves the review process to a higher level of abstraction since the development team is now reviewing general mathematical preconditions and postconditions instead of pages of code written in a programming language.

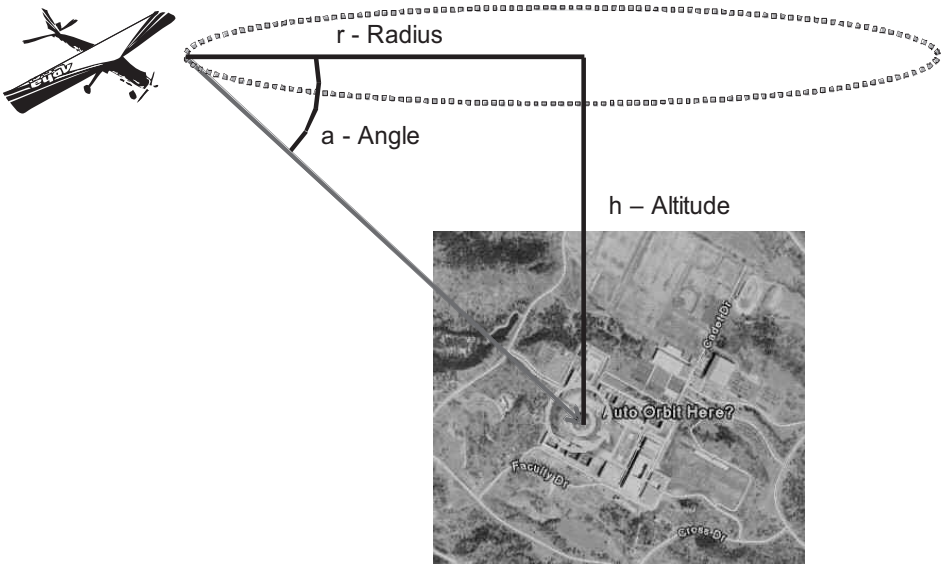
In the following section, we describe a small example that shows the SPARK development process in action. This example also highlights how the SPARK tools detect errors. At the U.S. Air Force Academy, we use SPARK to develop code in the senior-level software engineering course as we develop code to build flight plans for UAVs.

Figure 2: *Auto-Orbit Tool in UAVSAT*

UAV Situational Awareness Tool

In order to improve the situational awareness of a commander during a crisis situation, we have developed the UAV Situational Awareness Tool (UAVSAT) [7]. This year we have ported UAVSAT to Google Earth[2], which shows the location of the UAV in three dimensions along with a near real-time video feed from the UAV. The UAVSAT tool also provides the commander the ability to select a location on the map for the UAV to orbit. Figure 2 shows how the commander selects a location on Google Earth and indicates where the UAV should orbit. To position the UAV and gimbaled camera, the software we developed automatically calculates the optimal orbit altitude and radius to position it in the correct orbit to *stare* at the location selected by the commander. The software builds a new flight plan for the UAV in order to transition it from its current latitude and longitude to an optimal orbit radius and altitude.

Figure 3: *Calculating the Auto Orbit*



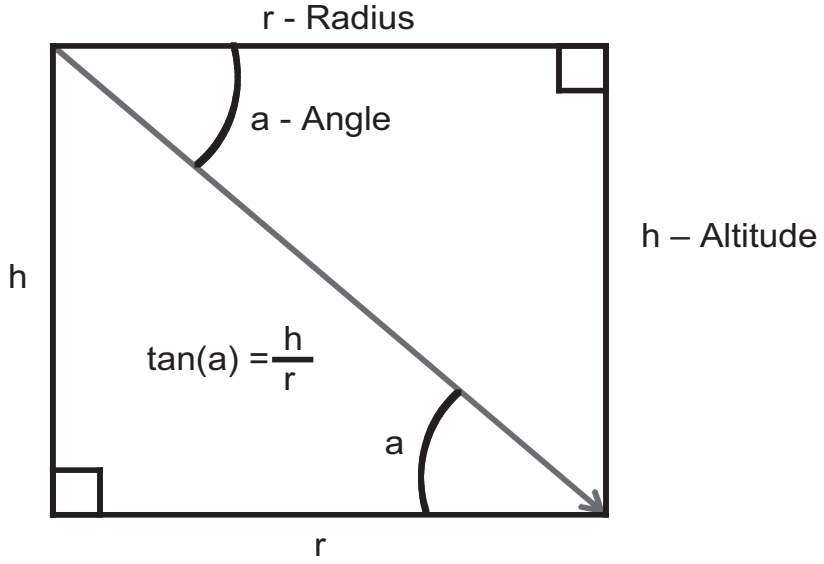


Figure 4: *Mathematical Depiction*

Using SPARK for Flight Control

Since the orbit flight plan for the UAV is built automatically by UAVSAT, it is imperative to calculate the correct altitude and radius for the UAV's orbit. If these values are not calculated correctly, the UAV will not be able to position the camera properly to observe the area of interest. If these calculations include flight critical phases, such as terrain avoidance or aircraft spacing, errors in the calculations could cause loss of the aircraft or destruction of objects on the ground. In order to ensure that the auto-orbit calculations are coded properly, we are using SPARK to verify the implementation.

Figure 3 (see page 11) shows which calculations need to be done in order to determine the proper flight plan for the UAV. In the figure, *h* is the altitude of the orbit, *r* is the radius of the orbit, and *a* is the angle of the gimbaled camera. The latitude and longitude of the orbit location are provided to UAVSAT as selected through the auto-orbit tool and positioned by the commander. To simplify the orbit problem, we have initially fixed the altitude of the orbit at 500 feet above ground level and have fixed the gimbal's angle at 30 degrees. The code for calculating the auto-orbit flight plan must simply determine the radius *(r)*, given the altitude and the gimbal angle.

Figure 4 depicts the problem as two similar right triangles. The tangent of the angle *(a)* is equal to the opposite side of the triangle over the adjacent side of the triangle. In the figure, this is represented by $tan\ (a) = h/r$. Solving for *r*, the result is $r = h/tan\ (a)$. That is, to calculate the radius of the orbit, we simply divide the altitude by the tangent of *a*.

In order for UAVSAT to receive the current latitude and longitude of the UAV and also to upload new flight plans to the UAV, our software must interface with C++ code provided by the autopilot vendor. We could have implemented UAVSAT in C/C++, but we wanted to preserve our ability to use the automated verification provided by SPARK. We therefore used Ada to build a Dynamically Linked Library (DLL) called from the C++ code. This DLL interface utility is well documented in the Ada Language Reference Manual[3].

Figure 5 shows the Ada_Radius function built in the DLL interface to calculate the radius *r* for the auto-orbit utility. Now the developer annotates preconditions and postconditions for the Ada_Radius function.

Figure 6 shows the specification of the Ada_Radius function. The specification includes the annotations for the preconditions and the postconditions. For functions in SPARK, the postconditions are annotated by using the *return* annotation [1]. The precondition for Ada_Radius allows heights greater than or equal to zero. The input angle is restricted from zero to avoid a potential division by zero error. The postcondition for Ada_Radius expresses the mathematical formula for the radius calculation. The angle is multiplied by Pi over 180 to convert the angle from degrees into radians as expected by the tangent function.

Figure 7 shows the body of the Ada_Radius function. Since the preconditions and postconditions are defined in the specification, no further annotations are needed in the body. The code is now ready to be analyzed by the SPARK Examiner. The developer executes the Examiner tool passing in the Ada_Radius function to be analyzed. The Examiner returns the verification conditions, and the developer executes the Simplifier, which returns the simplified verification conditions shown in Figure 8.

In the figure, the lines beginning with *H* represent *hypotheses* and the line beginning with *C* represents a *condition*. The symbol ->indicates *implication*. In this verification condition, H1 and H2 are derived directly from the precondition of Ada_Radius. H3 is determined during the process of hoisting the postcondition through the code and states that the result of the tangent function is restricted from zero. Since the radius is given as $h/tan(a)$, H3 is avoiding a potential division by zero. C1 indicates the final part of the verification condition built from the hoisting process. The code takes the first line of C1, and the postcondition takes the second line of C1. In order to prove the code is correct, we must show that this implication is true (i.e. that H1-H3 imply C1). So far, the developer has built the code and the annotations. The Examiner and Simplifier have built this simplified verification condition automatically for the developer.

The next step is to attempt to build a proof that this verification condition is true. The developer executes the Proof

Figure 5: *Ada_Radius*

```
-- function to calculate the radius in Ada
function Ada_Radius (
   Height : in   Float;
   Angle : in   Float)
 return Float is
begin
 return (Height / Ada_Tangent((3.14159/180.0)) * Angle);
end Ada_Radius;
```

Figure 6: *Specification for Ada_Radius*

```
-- function to calculate the radius in Ada
function Ada_Radius (
      Height : in  Float;
      Angle : in  Float)
   return Float;
--# pre (Height >= 0.0) and (Angle /= 0.0);
--# return (Height/Ada_Tangent((3.14159/180.0) * Angle));
```

Checker passing in the simplified verification condition. In this example, the Proof Checker is not able to build a proof and the developer must consider the possibility that the code is incorrect. Careful inspection of C1 shows that the *angle* variable has been moved to the front of the expression during the simplification process. This inadvertently highlights a discrepancy between the code and the postcondition.

Figure 9 shows the original Ada_Radius specification and body. Note the parenthesis on line 14 of Figure 9 for the call to Ada_Tangent. The postcondition formula on line 7 includes parentheses around the Angle variable, but the code implementation does not. This is a simple error in parentheses. This error is discovered during development because the verification condition for Ada_Radius cannot be proven to be correct. SPARK has found an error during the development phase where it can easily be fixed. Had the error not been found until the testing or implementation phase, it would have proven more costly. This simple example illustrates how SPARK reduces the cost of software development by finding errors during the development phase.

Figure 10 shows the corrected Ada_Radius code. The parentheses have been correctly placed around the Angle variable. Now when the program development team executes the SPARK tools on the code, a proof can be built that shows the verification condition is true. SPARK is able to automatically prove this code is a correct implementation for the preconditions and postconditions.

This simple example shows the power of the SPARK correctness by construction methodology. Even on a simple example such as this, an error in the code was discovered. Students in a senior-level software engineering course developed this example code. They eventually noticed the error in their code during testing and corrected it in a later version. Had they been using the SPARK approach from the start, they would have found the error while constructing the code and delivered correct code the first time.

Conclusion

As we have seen, the SPARK approach to constructing code is a powerful way to prove that the code being developed is a correct implementation given the preconditions and postconditions. This approach is now being used to develop safety critical flight control software for UAVs at the U.S. Air Force Academy as part of a UAVSAT that is designed to enhance a commander's ability to respond to dynamic situations effectively. The added benefit of using the SPARK approach to software development is that it helps assure the system is correct by construction.◆

```
-- function to calculate the radius in Ada
function Ada_Radius (
      Height : in Float;
      Angle : in Float)
   return Float is
begin
 return (Height / Ada_Tangent((3.14159/180.0)) * Angle);
end Ada_Radius;
```

Figure 7: *Body of Ada_Radius*

```
function_ada_radius_3.
H1:  height >= 0 .
H2:  angle <> 0 .
H3:  ada_tangent(314159 / 100000 / 180) <> 0 .
    ->
C1:  angle * (height / ada_tangent(314159 / 18000000)) =
    height / ada_tangent(angle * (314159 / 18000000)) .
```

Figure 8: *Results of Simplification for Ada_Radius*

References

1. Barnes, John. High Integrity Software: The SPARK Approach to Safety and Security. London, UK: Addison-Wesley, 2003.
2. Croxford, Martin, and James Sutton. "Breaking Through the V and V Bottleneck." Lecture Notes in Computer Science. 1031 (1996).
3. Croxford, Martin, and Dr. Roderick Chapman. "Correctness by Construction: A Manifesto for High-Integrity Software." CROSSTALK, Dec. 2005 <www.stsc.hill.af.mil/crosstalk/2005/12/index.html>.
4. Amey, Peter. "Correctness by Construction: Better Can Also Be Cheaper." CROSSTALK, May 2002 <www.stsc.hill.af.mil/crosstalk/2002/05/index.html>.
5. Shaw, M. "Abstraction Techniques in Modern Programming Languages." IEEE Software Oct. (1984): 10-26.
6. Guttag, John V., and James J. Horning. Larch: Languages and Tools for Formal Specification. New York, NY: Springer-Verlag, 1993.
7. Sward, Ricky, Tim Beerman, and Clint Sparkman. "Unmanned Eyes in the Sky." Military Geospatial Technologies 3.3 (2005).

Notes

1. Retrieved from Praxis High-Integrity Systems <www.praxishis.com> on Apr. 24, 2006.
2. Retrieved from Google Earth, a 3-D Interface to the Planet <http://earth.google.com> Apr. 24, 2006.
3. Retrieved from Ada Language Reference Manual <www.adahome.com/rm95/>.

Figure 9: *Original Ada_Radius Specification and Body*

```
1  -- function to calculate the radius in Ada
2  function Ada_Radius (
3     Height : in   Float;
4     Angle : in   Float)
5   return Float;
6  --# pre (Height >= 0.0) and (Angle /= 0.0);
7  --# return (Height/Ada_Tangent((3.14159/180.0) * Angle));

8  -- function to calculate the radius in Ada
9  function Ada_Radius (
10    Height : in   Float;
11    Angle : in   Float)
12  return Float is
13 begin
14   return (Height / Ada_Tangent((3.14159/180.0)) * Angle);
15 end Ada_Radius;
```

Figure 10: *Corrected Ada_Radius Code*

```
-- function to calculate the radius in Ada
function Ada_Radius (
   Height : in   Float;
   Angle : in   Float)
 return Float is
begin
 return (Height / Ada_Tangent((3.14159/180.0) * Angle));
end Ada_Radius;
```

About the Authors

Lt. Col. Ricky E. Sward, Ph.D., USAF, retired from the Air Force as an Associate Professor of Computer Science at the U.S. Air Force Academy. He retired as the Deputy Head for the Department of Computer Science and the Course Director for the senior-level two-semester Software Engineering capstone course. Sward received his doctorate in Computer Engineering at the Air Force Institute of Technology in 1997 where he studied program slicing and reengineering of legacy code.

Department of
Computer Science
2354 Fairchild DR STE 6G101
USAF Academy, CO 80840
E-mail: ricky.sward@wavmax.com

Lt. Col. Mark J. Gerken, Ph.D., USAF, is an Assistant Professor and the Deputy Department Head for Technology in the Department of Mathematical Sciences at the U.S. Air Force Academy. He currently teaches probability and statistics and has taught calculus, differential equations, and engineering mathematics. Gerken received his doctorate in Computer Engineering at the Air Force Institute of Technology in 1995 where he studied software architecture and formal program development.

Department of
Mathematical Sciences
2354 Fairchild DR STE 6D112
USAF Academy, CO 80840
E-mail: mark.gerken@usafa.af.mil

2nd Lt. Dan Casey, USAF, graduated from the U.S. Air Force Academy in May 2006 where he majored in computer science with an emphasis in information assurance. Casey is currently stationed at Ramstein AFB, Germany where he works as a communications and information officer.

435th Communications Squadron
Ramstein AB, Germany
E-mail: daniel.casey@
ramstein.af.mil

Web Sites

BuildSecurityIn

http://BuildSecurityIn.us-cert.gov

As part of the Software Assurance program, BuildSecurityIn (BSI) is a project of the Strategic Initiatives Branch of the National Cyber Security Division (NCSD) of the Department of Homeland Security. The Software Engineering Institute was engaged by the NCSD to provide support in the Process and Technology focus areas of this initiative. The Software Engineering Institute team and other contributors develop and collect software assurance and software security information that helps software developers, architects, and security practitioners to create secure systems. BSI content is based on the principle that software security is fundamentally a software engineering problem and must be addressed in a systematic way throughout the software development life cycle. BSI contains and links to a broad range of information about best practices, tools, guidelines, rules, principles, and other knowledge to help organizations build secure and reliable software.

Software Assurance Technology Center

http://satc.gsfc.nasa.gov/

The Software Assurance Technology Center (SATC) was established in 1992 as part of the Systems Reliability and Safety Office at NASA's Goddard Space Flight Center (GSFC). The SATC was founded with the intent to become a center of excellence in software assurance, dedicated to making measurable improvement in both the quality and reliability of software developed for NASA at GSFC. SATC is self-supported with internal funding coming from research and application of current software engineering techniques and tools. Research funding primarily originates at NASA headquarters and is administered by its Software Independent Verification and Validation Facility in Fairmont, WV. Other support comes directly from development projects for direct collaboration and technical support.

National Institute of Standards and Technology (NIST) Computer Security Division (CSD)

http://csrc.nist.gov/

The CSD-(893) is one of eight divisions within Information Technology Laboratory. The mission of NIST's Computer Security Division is to improve information systems security by raising awareness of information technology (IT) risks, vulnerabilities, and protection requirements, particularly for new and emerging technologies. NIST researches, studies, and advises agencies of IT vulnerabilities and devising techniques for the cost-effective security and privacy of sensitive federal systems; developing standards, metrics, tests and validation programs to promote, measure, and validate security in systems and services, to educate consumers, and to establish minimum security requirements for federal systems; and developing guidance to increase secure IT planning, implementation, management and operation.

CERIAS

www.cerias.purdue.edu

CERIAS is currently viewed as one of the world's leading centers for research and education in areas of information security that are crucial to the protection of critical computing and communication infrastructure. CERIAS provides multidisciplinary approaches to problems, ranging from technical issues (e.g., intrusion detection, network security, etc) to ethical, legal, educational, communicational, linguistical, and economical issues, and the subtle interactions and dependencies among them.

Application and Evaluation of Built-In-Test (BIT) Techniques in Building Safe Systems

James A. Butler
L-3 Communications

The use of automated functions to monitor and control the activities of potentially hazardous systems is almost unlimited. Ensuring an adequate built-in-test (BIT) (i.e., that a system is fully functional and operational) is one of the most critical aspects of developing safe, automated systems. However, because of its very specialized nature, application, and evaluation, BIT requirements and techniques are often neglected or misunderstood. This article presents some of the goals and uses of BIT, as well as the applications in providing a safe system.

In the early 1960s, the National Aeronautics and Space Administration (NASA) was building the Saturn V spacecraft to go to the moon and back. In order to accomplish this mission, NASA had to develop an automated navigation system. The hardware and software required for this journey was housed in an instrumentation unit (IU). However, no safety-critical software had ever been developed, and the use of computers and sensors to relay safety-critical data had never been used. Digital computers were in their infancy and the hardware was unreliable. In order to compensate, NASA built three identical units, each with its own hardware and sets of input sensors. The outputs from the three units were compared and if one was out of line with the other two, that output was ignored[1].

In today's world, computers are several orders of magnitude more reliable and are much faster. As a result, the use and complexity of tasks assigned to computational systems are also orders of magnitude greater. Today's processors and software programs are used in safety-critical applications, ranging from heart monitors to navigation systems to control of nuclear power plants and weapons. Development of full, independent redundant systems for most of these applications would be cost-prohibitive. As a result, systems and software engineers should consider applying built-in-test (BIT) procedures to the development of all safety-critical functions, determining where failures are likely to occur and what the effect of the failures will be on overall safety, along with providing backup procedures to allow safe task completion.

Although BIT evaluations are common, their use is often limited to a comparatively small number of software and system developers as BIT is not usually taught as a formal curriculum in schools. As a result, BIT may not be considered by systems engineers for new systems, particularly when the purpose of the new system is to demonstrate a new capability or engineering concept. The assumption is that if the design is good and thoroughly tested, all of the hardware/software functions will perform as intended. This article provides the reader with a basic understanding of how BIT techniques are used in embedded systems and how they can be incorporated into new system requirements to reduce the risk of unanticipated system failures. With this understanding, the reader (systems engineers, hardware developers, or software developers) should have at least some capability to evaluate the effectiveness and completeness of BIT requirements and functions employed in safety-critical systems.

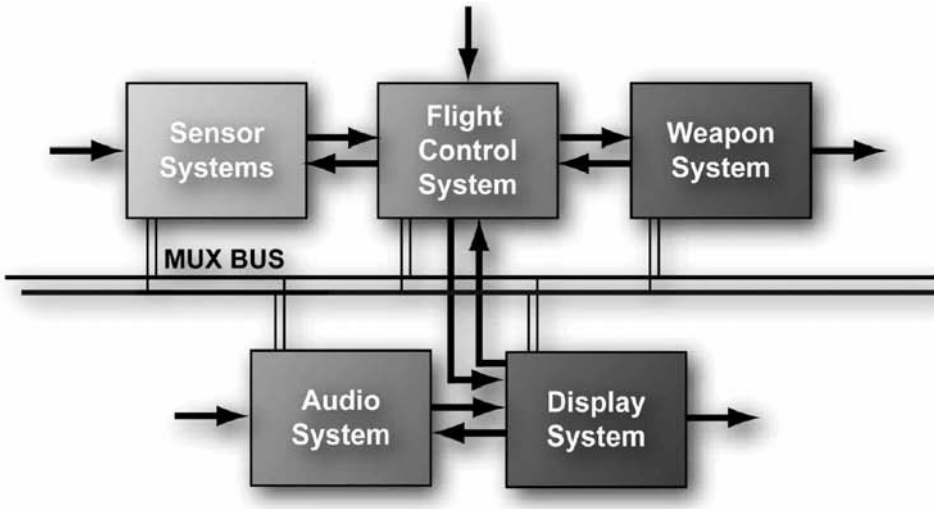
The backup procedures and responses to anticipated errors should also be considered in developing a safe system. The development of backup procedures is unique to each application and is beyond the scope of this article. However, backup sets of inputs provide a partial redundancy at a much lower cost than a full system redundancy. These backup sets of inputs may include additional sensors used to verify the primary sensor inputs. For instance, velocity of an aircraft can be measured by wind velocity or with Global Positioning System (GPS) inputs. The wind velocity is considered more accurate, but the GPS has more checks and may be more reliable; therefore, the GPS inputs may act as the backup for the wind velocity measurements. Alternate procedures, such as activation of a backup system or initiation of manual procedures when the automated system(s) fails, should also be included in the system engineer's requirements.

Typical Embedded System

In order to understand BIT requirements for an embedded system, one has to understand how an embedded system operates. Figure 1 presents a simple system consisting of several subsystems. During normal operation, each subsystem provides and receives data that are critical to performing safety-critical functions. If a sensor system fails to provide the correct data to the flight control system or the data is corrupted during transmission, the safety of the aircraft and passengers could be jeopardized.

Each subsystem may also be comprised of lower-level components, as shown in Figure 2 (see page 16). Each component has the potential of corrupting the subsystem as well as other subsystems in the system. In the example subsystem, Processor 1 reads inputs from var-

Figure 1: *System of Subsystems*



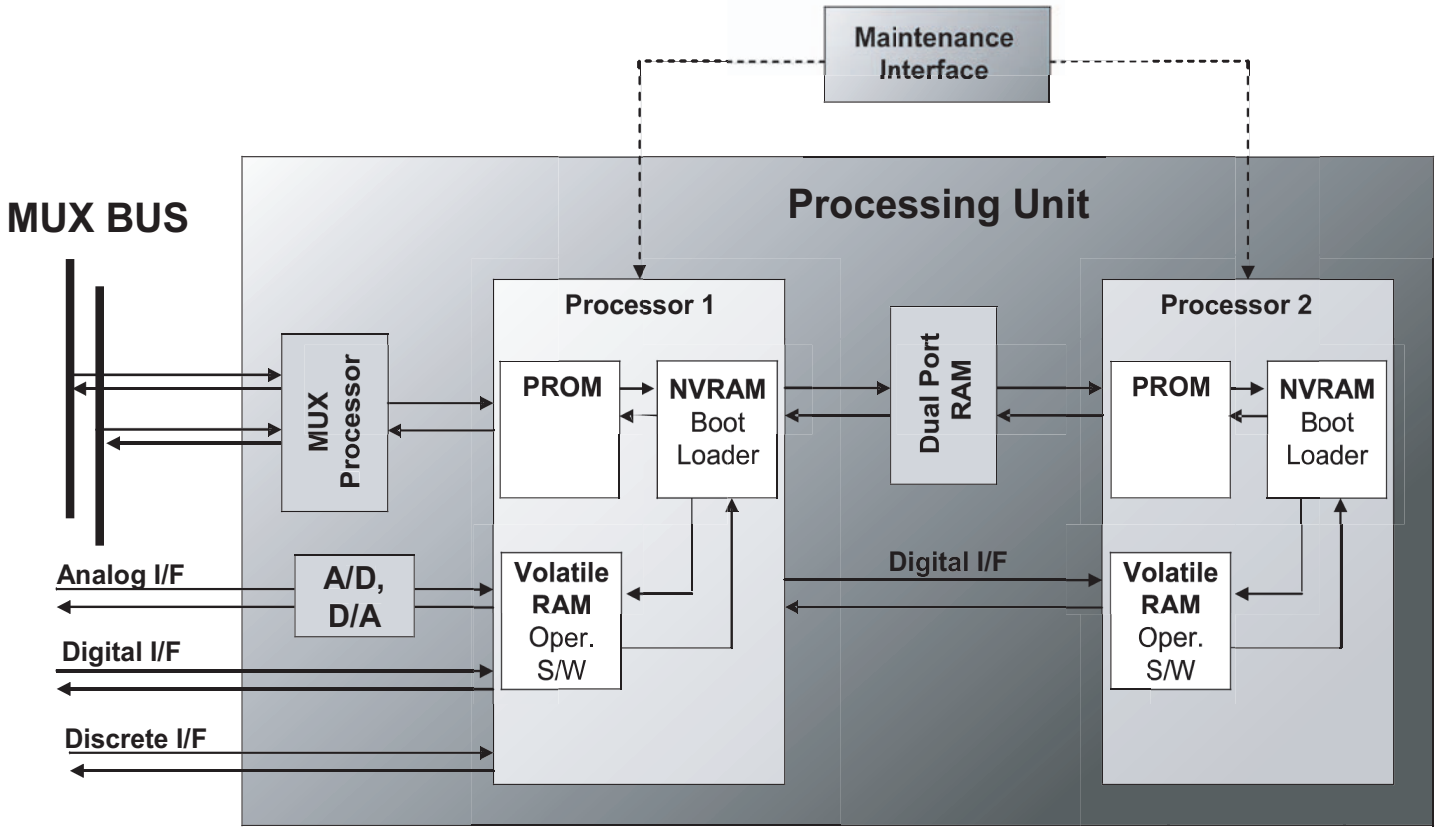


Figure 2: *Typical Embedded System/Subsystem*

ious sensors, user responses, etc; reformats the inputs into engineering units; and provides the data to Processor 2. Processor 2 determines control commands based on the inputs and provides responses to Processor 1. Processor 1 then reformats the outputs and sends the data to each of the respective output units where it may be used on other safety-critical functions.

Interfaces to the system may include operational interfaces such as multiplexer (MUX), digital, discrete, analog, and maintenance interfaces. Internal interfaces include all processor-to-processor and processor-to-peripheral interfaces, including dual-port random-access memory (RAM), which is memory shared by both processors.

The processors typically consist of programmable read-only memory (PROM), non-volatile RAM (NVRAM), and volatile RAM. PROM is used to store data and operational code while the system is not in use. The boot loader is usually installed and runs on NVRAM. During normal startup operations, the boot loader moves the operational code from PROM to volatile RAM and activates it. The volatile RAM is used to house the operational code and provide temporary storage for data during real-time operations.

Software and/or hardware interrupts are usually used to synchronize the activities between processors. In a data-driven system, Processor 1 provides an interrupt to Processor 2 when new data is available. Conversely, when Processor 2 is ready to output its data, it provides an interrupt to Processor 1. In a time-driven system, Processor 1 and/or 2 are provided with time-driven hardware and/or software-driven interrupts.

Operational Functions

In the typical system discussed here, there could be a number of high-level functions including the boot loader, operational code, and interface manipulations:

- **Boot loader.** The boot loader codes typically reside in and operate out of NVRAM. In a multi-processor system, there would probably be a boot loader for each processor. Since the boot loader operates out of NVRAM, the code is usually *burned-in* at the factory. As a result, all reprogramming of the code requires removal of the processor from the board; thus, functional requirements are kept to a minimum. The primary functions of the boot loader include, first, moving the operational code from PROM to RAM and initiating the operational code and, second, providing the capability to read the operational code from the maintenance interface and write it to PROM, thereby providing the capability to update the operational software and providing an access for performing maintenance evaluations on the processor(s).
- **Operational code.** The operational code is responsible for performing the operational functions of the system. After the operational code has been written in RAM, it has full control of the system until the system is powered down or has its authority removed by another system.
- **Interfaces.** The interfaces provide access to data from other systems as well as intra-system data and include MUX, analog to digital (A/D) and digital to analog (D/A), discrete, and dual-port RAM interfaces.

Modes of BIT and Their Functions

Generally, there are four modes of BIT: Startup BIT (SBIT), Continuous BIT (CBIT), Initiated BIT (IBIT), and Maintenance BIT (MBIT).

- **SBIT** is used in the evaluation of key functions and capabilities before the start of the application. SBIT is usually performed or at least initiated by the boot loader and provides a GO/NO-GO response to the system and users. Some of the test functions performed in SBIT include memory, boot load,

and interface tests. Startup culminates in initiating the operational code.

- **CBIT** is run out of the operational code and is used to evaluate selected elements and functions during the mission. CBIT is especially applicable to non-expendable systems such as airplanes or weapons, where passenger or bystander safety is involved. The operational code generally performs foreground and background BIT tasks. The foreground tasks include all necessary activities to accomplish the primary operational tasks, including evaluations of inputs to the system. The background tasks include CBIT activities that cannot be performed in the foreground. The following are the activities performed in each:
  - o Foreground tests are the tests needed to assure that the inputs, processor, and software are valid. These tests are preformed in each interrupt cycle. Specific foreground tests include the following:
    - Input tests used in input verification include checksum; parity checks; time tags; sequence numbers; heartbeat checks of digital and discrete inputs and voltage, current, and frequency checks for analog and power inputs.
    - Output tests are used to provide the receiver with the ability to verify the accuracy of the digital message or verify that the analog/discrete outputs were correctly received.
    - Processor evaluations are used to evaluate the health of the processor. Specific tests include interrupt monitoring, evaluation of timing, and memory use.
    - Software evaluations include tests on the software to prevent inaccurate or out-of-bounds data from causing an irreversible error and evaluation of the exception handling procedures provided in some software languages.
  - o Background tests are performed on an *as available* basis. Generally, the background tests require more time to complete than what is available from the system. As a result, background tests are performed after operational code has completed its operational functions. Background tests include the following:
    - Input tests performed in the background including loopback tests of digital, discrete, and analog inputs, as well as non-destructive Stuck-on-1/Stuck-on-0 tests of interface buffers.
    - Memory tests including non-destructive Stuck-on-1/Stuck-on-0 tests of all applicable memory locations.
- IBIT is a detailed BIT used in mission readiness assessments or to assist MBIT in pinpointing sources of errors in the system. Typically, IBIT would be used to pinpoint faults in the system down to the line replaceable unit (LRU) level. IBIT is almost always applicable to non-expendable systems but may also be used in expendable systems such as a *smart weapon*. Since IBIT overrides the operational code functions, it is not used while the system is performing its normal functions.
- MBIT is exhaustive BIT designed to interface with the maintenance port on the unit. A maintenance port is usually provided in the hardware design, specifically to allow for maintenance on the unit. This provides *peek and poke* capabilities into designated memory locations. The more common use is the capability to load new software into the system without having to dismantle the unit or perform any hardware manipulations. MBIT is applicable to all systems as a development platform and a provider of software upgrades.

BITs to Be Performed

Complete evaluation of a system requires several BIT functions. The following is a summary of the BIT functions usually performed in a safety-critical embedded system:

- Download BIT verifies that the software being loaded into PROM from the maintenance port is correct. Download BIT is performed by the boot loader software as part of MBIT.
- Boot Load BIT verifies that the operational software being loaded from PROM to NVRAM is correct. Boot Load BIT is performed in SBIT.
- Memory test verifies that the memory locations (i.e., RAM, scratch pad, working areas) are functioning correctly. Memory tests may be performed in all modes of BIT.
- Interface BIT verifies that the data being passed across the interfaces are correct. Interface tests are used in all modes of BIT but are more critical in CBIT.
- Power BIT verifies that the power inputs are within defined tolerances. As such, Power BIT is used in all modes of BIT.
- Processor BIT verifies the functionality of the processor in real-time. Its primary use is in the CBIT mode.
- Software BIT verifies the operational functionality of the software and is part of CBIT.

BIT Algorithms

There are several ways to perform each of the BITs. The following are some of the more widely used techniques:

- *Checksum* is a sum of all words within a designated memory location and is used to verify inputs to the system or to verify input data matches the data sent or that the data in a memory block has not been corrupted. In performing a checksum, the sum of all words in a message or memory block is written to a designated location. In validating the data block, the data within the block are summed and compared to the original checksum; if the two checksums do not agree, the test fails.
- *Parity* is a single bit sum of all bits within a word or memory location(s) and is used in evaluation of discrete ON/OFF inputs. A typical example is a bank of toggle switches controlling the functions of the software system. An exclusive OR (XOR) is used for the evaluations:

  Parity = Switch_1 XOR Switch_2 XOR . . . XOR Switch_n

  Generally, an odd parity (i.e., XOR an additional 1 to the function) is preferred, as loss of power to the bank of toggle switches is apparent: all zeroes, including the parity, indicate an error such as loss of power to the toggle bank and a 1 indicates that all switches are OFF and valid, and the bank of toggle switches have power.
- *Stuck-on-1/Stuck-on-0* provides the ability to evaluate all bits within a word or memory location to ensure its ability to maintain both 1 and 0. Data patterns that exercise all bits such as 0x55 (01010101) and 0xAA (10101010) are written to the memory location and read back. If the pattern read from the memory location does not match the one written, the test fails. In a non-destructive test, the original value in the memory location is saved and restored once the test is complete.
- *Performance BIT* monitors the processor speed and availability as part of the processor BIT.
- *Time tagging* allows the evaluation of

Purpose:
- Provide non-intrusive monitoring of equipment, interfaces, etc., to ensure that they are operating within measurable limits.
- Provide information to the software for evaluation of the system's capability to perform.

Examples:
- Voltage and current meters to evaluate interfaces, power inputs, etc.
- Frequency monitoring of analog interfaces.
- Monitoring of peripheral equipment activities.

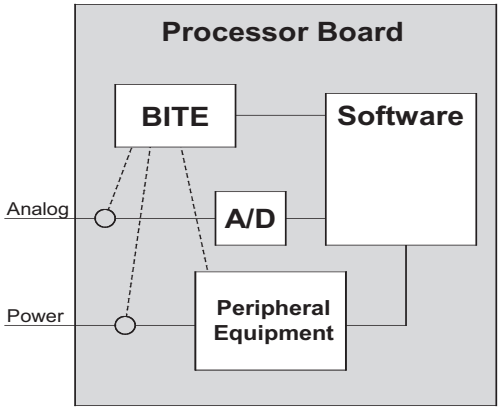


Figure 3: *Examples of Built-In-Test Equipment Functions*

the time that a message was sent or the occurrence of an event with respect to its last occurrence. Time tagging is used most often in validating input messages or as assistance to the receiving processor of another subsystem.

- *Sequence numbering* provides the ability to evaluate input messages with respect to the last message: A 1 is added to the sequence number for each new message by the sender and verified by the receiver.
- *Loop back tests* evaluate interfaces by sending a message to the interfacing processor or analog device which sends the message back. The original message is compared with the message that was looped back.
- *Memory usage* evaluates the available memory during normal use.
- *Interrupt testing* provides evaluation of the interrupt routines and functions.
- *Exception handling* is a pre-programmed reaction to run-time errors. Its applicability in BIT is to assure an adequate reaction to a processor or software error.
- *Stack overflow* is a software test used to evaluate arrayed variables to ensure that they do not overflow the array dimensions and overwrite other data.

BIT Equipment (BITE)

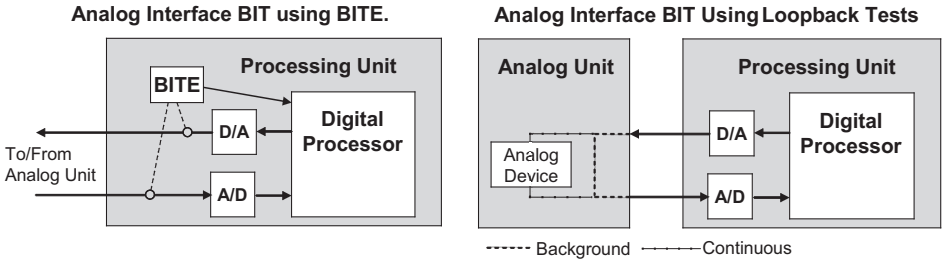
Figure 3 presents an overview of how BITE is used in a processor system. The primary purpose of BITE is to provide a non-intrusive analysis of the systems or interfaces that cannot be directly evaluated using conventional software BIT procedures. The operational code uses the inputs from BITE as a GO/NO-GO indication of the health of the unit or interface being tested.

Typical Test Procedures

Test procedures used in BIT are usually a combination of several BIT techniques or algorithms. The following items present an overview of some of the more common test procedures:

- **Download.** Download of software is typically used to update the operational code stored in PROM. This allows the developer to upgrade the software in the field without having to remove any hardware from the system. The new software is usually read in from the maintenance port and written to PROM. The format for the download code usually includes a header, the text of the code, and a footer. The header includes the number of words in the record, a sequence number for the record, and the function to be performed in the download process. The text contains the operational code and the footer contains the checksum of all words in the record. The procedures used to validate the code include the following:
  - **Sequence checking.** The *sequence checking* ensures that no record is skipped or read twice.
  - **Record checksum.** The *record checksum* is used to validate that each record is correctly read and stored in PROM. Each word in the record is written to PROM, read back, and summed with the rest of the words in the record and compared with the checksum in the footer.
  - **Block checksum.** The *block checksum* is the final download test and is used to make sure that all procedures were done correctly. The final word in a code block is typically a checksum of all words in the block of code. The final validation check is made by re-reading all of the memory in the designated area of PROM and performing a checksum on each word.
- **Boot load.** The procedures for boot load are similar to download; except, time is usually more critical. The data is read from PROM and written to and read back from RAM. A continuous checksum is performed on the data read back from RAM. When all data has been written to RAM, the running checksum is compared with the block checksum in PROM. If the checksums agree, the boot loader initiates and transfers control to the operational code.
- **Memory.** Stuck-on-1/Stuck-on-0 tests are the most widely used methods for evaluating memory. Typically, the boot loader initiates the Stuck-on-1/Stuck-on-0 testing of RAM using a destructive test (i.e., the RAM memory will be cleared before the code is written). In CBIT, non-destructive testing is performed in the background.
- **Interfaces.** Without question, errors introduced in hardware to software and software to hardware interfaces provide the greatest opportunity for introduction of errors into the system. For this reason, the industry has developed some good, and in some cases, sophisticated BIT capabilities. The following presents an overview of the interface BIT capabilities:
  - **MUX Interfaces.** MUX interfaces, such as the MIL-STD-1553, are purchased as a standard protocol, along with the MUX processor hardware. The 1553 has been around since the early 1970s, primarily because of the BIT used in detecting errors and verifying the data being transferred[2]. Each message received via the 1553 has a header, body, and footer:
    - **Header.** The header provides the sub-address number, number of words in the record,

Figure 4: *Overview of BIT Procedures for Analog Interfaces*



sequence number, and a timestamp.
- **Body.** The body text contains the data being sent to the receiving processor.
- **Footer.** The footer contains a checksum of all data in the record.

The following presents an overview of the tests used by the 1553 to validate the correctness of the data:
- The sequence number is evaluated, ensuring that no messages were lost or that no duplicate was sent.
- The timestamp is compared to the previous timestamp to ensure updates are being received as designed.
- The data is check-summed and compared with the checksum in the footer.
- The MUX BUS provides *heartbeat* messages to indicate it is operational, even when no messages are being sent.
- If any of the previous tests fail, a signal is passed to the operational code so that backup procedures can be activated.

It should be noted that all communication protocols use similar error-checking procedures. Most of the digital audio and visual protocols also contain error correcting routines.

o **Processor-to-processor and dual-port RAM interfaces.** In transferring data from one processor to another, the sending processor writes to a designated memory address used by another processor. Some of the *off-the-shelf* hardware packages such as dual-port RAM have error detecting and error preventing code built into the system. Additional error detecting capabilities are easily included in the BIT process and include sequence numbering, timestamps, heartbeat, and especially checksumming as described in the MUX interfaces.

o **Analog interfaces.** Analog devices are usually very dumb; by their very nature, they do not provide any way of evaluating the input or output signals. Therefore, errors are common from the analog sensor itself or during transmission. Analog interfaces include both inputs and outputs to the system. In order for the digital processor to be able to read analog inputs, an A/D processor converts the analog signal to its digital counterpart. Conversely, the D/A provides the analog device with the electrical signal corresponding to the digital command from the digital processor. Validation of input signals is typically performed by the following:
  - **Range checking the signal.** In order for range checking to be effective, the full electrical range of the analog device must not be used for normal inputs. Thus, a *zero* or *full scale* reading is distinguishable as an error and not a normal input.
  - **Comparing the signal with redundant analog signals.** The use of redundant signals is a highly reliable method of detecting errors in analog inputs, assuming that all *common cause* errors, such as common grounding, have been eliminated. One way of enhancing the reliability of the redundant signals is to set the signals in opposite directions – sensor A provides inputs from low scale to high and sensor B provides inputs from high to low.
  - **Reasonableness checks.** Reasonableness checks evaluate the data to ensure the inputs are reasonable, given the previous data. If the sensor provides inputs that are physically impossible (i.e., the velocity goes from +100 MPH to -100 MPH in a single 0.01 sec. cycle), the sensor data is probably incorrect and should not be used in making critical control commands.

  Validation of output signals are usually performed by use of BITE or loopback tests, as presented in Figure 4.
  - BITE monitors the analog input and output channels to ensure the correct range of voltage, current, and/or frequency signals are provided.
  - BITE can also be used to perform an internal loopback test which can be used to validate the A/D and D/A converters within the processing unit.
  - Loopback tests that do not engage the analog device are performed in SBIT or background CBIT.
  - Continuous foreground CBIT loopback tests evaluate each signal to the analog device. Since the foreground CBIT evaluates all command signals, it is the preferred method of BIT evaluation for analog outputs.

o **Discrete interfaces.** Discrete interfaces are typically used to activate (turn on) or deactivate (turn off) a particular function or input. Figure 5 provides an overview of testing discrete inputs. Parity evaluations provide evaluation of all inputs at all times and is considered superior when compared to other methods of evaluations, such as loopback tests, which must be performed in the background.

- **Processor.** Evaluation of processors is particularly important as new code is added or existing code has been modified. Some of the processor BITs include the following actions:
  o Evaluate the performance of the system by, first, setting time limits on hardware to software functions and measure the time to perform primary functions; and, second, setting flags to determine if any high-priority functions are not performed in an interrupt cycle.
  o Measure and compare the memory usage with threshold percentages to ensure adequate capability during operations.
  o Evaluate timed interrupts by providing hardware interrupts (i.e., watchdog timers) to evaluate software functions and vice versa.
- **Software.** The purpose of software

Verification of the discrete interface can be performed with:
- Parity checks:
  - Each discrete input is input to Exclusive OR (XOR) logic.
  - The processor performs an XOR of the discrete inputs and compares it to the one received from the discrete source.
  - Odd parity is recommended because a series of FALSEs (0) would provide a TRUE (1) in the parity bit (i.e., if the unit were off-line, all bits would be False).
- Loopback tests with the other unit, evaluating both TRUE and FALSE conditions.

Discrete Interface BIT using Parity checks.

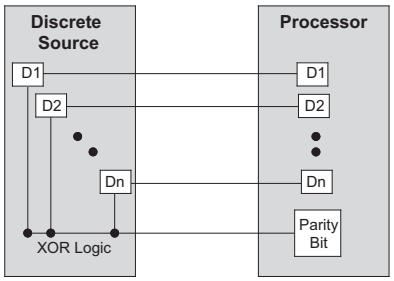


Figure 5: *Evaluation of Discrete Inputs*

Coming Events

October 2-3
The DoD-DHS Software Assurance Forum
McLean, VA
https://buildsecurityin.us-cert.gov/daisy/bsi/events.html

October 9-11
CNIS 2006 Communication, Network, and Information Security
Cambridge, MA
www.iasted.org/conferences/2006/cambridge/cnis.htm

October 10-11
PNSQC 2006
The 24th Annual Pacific Northwest Software Quality Conference
Portland, OR
www.pnsqc.org

October 10-11
VERIFY 2006
International Software Test Conference
Washington D.C.
www.effectivesoftwaretesting.com/conference_verify.aspx

October 16-20
STAR WEST 2006
Software Testing Analysis and Review
Anaheim, CA
www.sqe.com/starwest

October 23-25
MILCOM 2006
Military Communications Conference
Washington D.C.
www.milcom.org/index.htm

October 23-26
SEC 2006
9th Annual Systems Engineering Conference
San Diego, CA
www.ndia.org

2007
2007 Systems and Software Technology Conference


www.sstc-online.org

BITs is not to perform validation of the software but to monitor its functions and inputs to ensure that the software does not crash during critical operations. The procedures available for software evaluations include the following:

- **Data analysis.** First, perform sanity checks on input data and, second, prevent run-time errors (divide by zero, log of zero or a negative number, etc.) by ensuring that incorrect and out-of-bounds data are not used.
- **Stack overflow.** Provide software checks to ensure against and report conditions where stacks overflow (especially necessary in C, C++, and other languages).
- **Exception handling.** Provide exception handling capabilities in the code development so that run-time errors do not cause the system to crash. Ada and other languages provide for exception handling routines.

Confirmation of Emergency Procedures

Providing responses to emergency situations is one of the primary responsibilities of safe software. Ensuring that the indicated response is correct is often dependent on knowing that all inputs, procedures, and commands are correct. To this end, the software system must provide adequate BIT *before* the emergency procedure is activated. The use of foreground rather than background tests reduces the time required to confirm that emergency procedures are required. If errors are detected, the backup or redundant inputs and procedures built into the system provide a safe alternative. If the data was not corrupted, a clear emergency procedure should be activated. If data errors are detected by BIT, the backup or redundant inputs and procedures built into the system provide safe alterations without having to activate emergency procedures.

Summary

The cost of evaluating BIT and implementing recommended improvements is relatively inexpensive when built into the system from the beginning. The ability to perform BIT on most interfaces is determined during the high-level system design; BIT between subsystems requires that both subsystems perform their respective BIT functions. However, correcting interface BIT oversights requires making changes to at least two software components and may also require modification of the hardware design.

There are no hard and fast rules concerning BIT – certainly no one size fits all recommendations. As a result, the application determines the amount and detail of BIT required. The number and type of responses to BIT are dependent on the criticality of the application and the likelihood of the system error. Each response criteria is determined by available software and system backups and the level of operational capability that is desired. As a result, each backup procedure needs to be evaluated for its adequacy in meeting the response criteria.◆

Notes

1. John Duncan presents a great overview of the development of the Instrumentation unit in <www.apollosaturn.com/s5news/p71-7.htm>.
2. Several companies provide very good Web sites to their 1553 products and full standard descriptions are available from the government. The PDF presented, <www.testsystems.com/pdf/overview.pdf>, provides a good overview of the development and use of the 1553.

About the Author

James A. Butler has more than 35 years in software safety analysis and software development, including embedded systems, system engineering, BM/C3, and engineering. He has performed software safety analysis on the Apache helicopter flight management computer. The analyses performed included requirements review, design implementation, and technical adequacy test; analysis of the design using software failure modes, effects, and criticality analysis; and determination of critical software components, inputs, and algorithms. Butler has worked closely with designated engineering representatives in performing software safety evaluations for the Chinook helicopter, digital advanced Flight Control System, and C-130 Intercommunication and Radio System, and he has evaluated applicable safety standards for the Army's Future Combat System.

4121 Hide-A-Way DR
Guntersville, AL 35976
Phone: (256) 505-0808
E-mail: jimjudybutler@bellsouth.net

Assessing Information Security Risks in the Software Development Life Cycle

Douglas A. Ashbaugh
Software Engineering Services

Information is among the most important assets in any organization. Organizations are constantly building more complex applications to help them accomplish their mission and are entrusting their sensitive information assets to those applications. But are their information assets secure as they are transmitted, modified, stored, and displayed by those applications? Are new applications developed in a manner that will keep those sensitive information assets secure? How can we know for certain? The answers to these questions are all related and involve the process of assessing risk.

The concept of risk – *the net negative impact of the exercise of a vulnerability, considering both the probability and the impact of occurrence* [1] – is a concept that is hundreds of years old. Even the concepts of measuring and weighing *business risks* have been around for a long time. Insurance companies calculate risks every day and use the calculations to set rates for life, health, and property coverage.

Software development is a constant balancing act between functional requirements, funding, deadlines, limited resources, risk, and flexibility. Many of the current major software development life cycles (SDLCs) treat security simply as just one more non-functional requirement [2] and do not cover the topic of information security or address it in any detail. The result is often that security remains a non-functional requirement during the software development process. During the software engineering process, when resources, budgets, and schedules become tight, trade-offs must be made as some requirements must be dropped. This trade-off process introduces risk into the software development process. This is not to imply that security is always an important requirement of every software development effort. However, if confidentiality, integrity, and availability of the software or the information it stores, transmits, processes, or displays is important, then security should be considered an important requirement.

When risk is introduced into the software development process where confidentiality, integrity, and availability of the software or its information are important, then the result may be that the resulting software is not as secure as it needs to be. The General Accounting Office estimates $38 billion per year [3] in U.S. losses due to costs associated with computer software security lapses. How can we resolve this problem?

One solution is to apply information security risk assessment practices to the SDLC. Information security risk assessment is a practice used to ensure that computing networks and systems are secure. By applying these methods to the SDLC, we can actively reduce the number of known vulnerabilities in software as it is developed. For those vulnerabilities that we cannot or choose not to mitigate, we at least become aware of the risks involved as software development proceeds. The remainder of this article will focus on how to apply simple risk assessment techniques to the SDLC process.

Assessing Risk Within the SDLC

There are many different methodologies, tools, and techniques that can be used to assess risk. For the purposes of this article, we will focus primarily on a simple *qualitative* method. *Qualitative risk* assessments are all about identifying and relating risks relative to each other. The perceived impact of a loss associated with a risk is determined rather than the actual value associated with the loss. Also, because they are subjective in nature and do not require the precise knowledge required by other risk assessment methodologies, they typically take less time to conduct.

There are a number of essential elements of any qualitative risk assessment process. First, assets, threats, and vulnerabilities must be identified. An analysis can then take place that determines the likelihood of a vulnerability being exploited, the adverse impact of a vulnerability being exploited, and, finally, the level of risk associated with each threat-vulnerability pair. Controls may then be applied to eliminate or prevent the exercise of vulnerabilities. From a high-level view, the relationships between these entities are outlined in Figure 1. Looking at each of these elements a little closer, some of the ways we can apply them in each phase of the SDLC become visible.

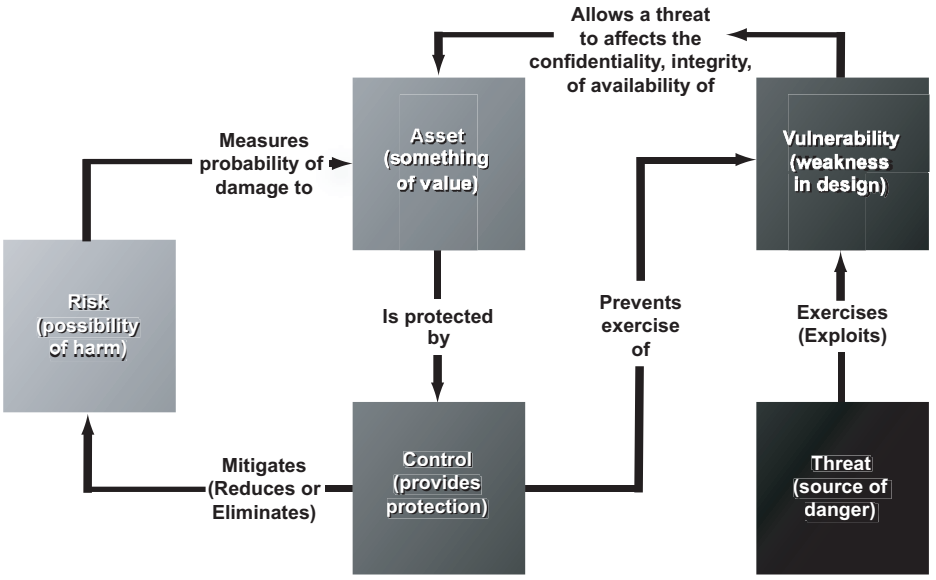
Identifying Threats

A *threat* is something that is a source of danger to an asset. Sources of threats may include (but are not limited to) those listed in Table 1 (see page 22).

The following organizations have published lists of potential threats:

- The National Institute of Standards and Technology (NIST) [4].

Figure 1: *Relationships Between Risk Analysis Elements*



Human Threats	Technical Threats	Environmental Threats
Data entry errors and omissions	Misrepresentation of identity	Electromagnetic interference
Inadvertent acts and carelessness	Unauthorized access to systems	Hazardous materials
Impersonation	Data contamination	Power fluctuations
User abuse and fraud	Malicious code	Water leaks
Theft, sabotage and vandalism	Session takeover	Fire
Espionage		Natural disasters

Table 1: *Threat Sources*

Checklist Sources	Application Code Scanning Tools	Network Vulnerability Scanners
The Open Web Application Security Project (OWASP) [8]	Paros Proxy	Nessus
Common Vulnerabilities and Exposures (CVE) [9]	WebScarab	Nmap
National Vulnerability Database (NVD) [10]	Web Inspect	Nikto
		Whisker

Table 2: *Identifying Vulnerabilities*

- The SysAdmin, Audit, Network, Security (SANS) Institute [5].
- The Center for Medicare and Medicaid Services (CMS) [6].

Many of these lists provide not only descriptions of threats, but also many real-world examples, as well as the potential impact a threat may have on the confidentiality, integrity, and availability of an asset.

It is important to identify threats because it is necessary to understand the potential impact a threat may have upon an asset. Once this potential impact is understood, controls that safeguard the asset from the threat can be identified. The earlier such controls are identified within the SDLC, the easier they can be incorporated into the design.

Identifying Vulnerabilities

A *vulnerability* refers to a weakness in design or controls that can be exploited by a threat to cause harm to an asset. According to the CERT Coordination Center, more than 90 percent of software security vulnerabilities are caused by known software defect types, and most software vulnerabilities arise from common causes. In fact, the top 10 causes account for about 75 percent of all vulnerabilities [7].

One of the easiest ways to identify vulnerabilities is to use checklists of common vulnerabilities developed by government and other interested agencies and groups. Another method would be to use application code scanning tools. Finally, network vulnerability scanners can also be used to find potential security vulnerabilities within an application as it exists in a network environment. A list of these checklists and tools is outlined in Table 2.

Identifying Assets

An *asset* is something of value to an organization. Assets associated with software development may include – but are not limited to – software, information, services, processes, functions, business rules, encryption keys, and methods.

It is important to identify assets as early as possible. Without knowing what assets need protection and without understanding what may happen when that protection fails, risk analysis techniques cannot produce worthwhile results [11].

In the case of identifying assets for software development, it is important to look to the business areas that will use the application being developed and its data. The end-users and their management are in the best position to understand what business information is essential to the mission and goal of the development effort. Furthermore, end-users and their management are in the best position to identify the business impact of the failure to protect the critical information.

Assets can also be identified by looking at policy. If an organization has good enterprise security policies, then assets can be discovered by a review of those policies to determine what types of assets the policy strives to protect. For example, a data classification policy might require that all customer information that could directly identify a customer (i.e. name, address, customer identification, etc.) must be considered confidential. If the application under development were to use or process any of this information, we would then consider that information an asset. Another example might include a disposal and reuse policy which requires that media must be thoroughly sanitized prior to its reuse or disposal. The assets in this case are hardware and/or reusable media storage devices and the information stored on them. Likewise, other policies may deal with the physical protection of people, hardware, documentation, buildings, and services.

Once information assets have been discovered, we can then apply analysis techniques to discover other potential assets. We can look at use cases, process flow diagrams, code, and other documentation and find where identified assets are accessed, read, written, modified, used, or monitored. Are specific processes, methods, services, functions, hardware, or individuals used to modify, display, transmit or store the assets? Any of these items may also be assets requiring protection.

Analyzing Risks

Risks occur when a threat exercises a vulnerability. The first step in analyzing risks is to determine the likelihood that a threat-vulnerability pair will be exercised. This is accomplished by consulting a likelihood table such as Table 3. Note that Tables 3-5 are provided as examples of what likelihood, impact, and risk tables might look like. Organizations such as NIST, SANS, CMS and others have developed a wide variety of tables that may be used in developing a table to fit your organization's specific needs.

Once the likelihood of a threat-vulnerability pair being exercised has been determined, then the impact to the assets identified if the threat-vulnerability pair is exercised may be determined according to Table 4.

Now the *risk* that a given threat may cause a specific impact by the exercise of a vulnerability may be determined. The *risk level* is determined by cross-referencing the likelihood with impact as shown in Table 5.

The process should be continued until all of the risks have been assessed. Once all risks have been assessed, management can prioritize, evaluate, and implement the most appropriate controls in order to mitigate the risks that have been uncovered. There are several options and strategies for mitigating risks including, but not limited to the following:

- **Risk assumption.** Choosing to accept the potential risk as is, or implementing controls to lower the risk to an acceptable level.
- **Risk avoidance.** Avoiding the risk by eliminating the cause or consequence of the risk.
- **Risk limitation.** Limiting the adverse affects of a risk by implementing additional controls.

- **Risk transference.** Transferring the risk to compensate for the loss, such as purchasing insurance.

Assessing Risks Within the SDLC

It is as important to know *when* to assess risks within the SDLC as it is to know *how* to assess risks within the SDLC. Risks should be assessed at the very beginning of the project and continue with each program review. It is also important to assess risks whenever requirements change and when the potential for new vulnerabilities and new threats are introduced.

Example of Risk Assessment

Consider an application for the financial services industry that requires a lot of personalized customer information such as names, employee identification numbers (which may be social security numbers), salaries, contributions to retirement plans, and dates of birth. Customer information for the application is gathered by a specific process and stored in a database. Complex business rules, which are proprietary to the organization, are used within the application to calculate individual retirement benefits. Furthermore, organizational policy requires that all information that could specifically identify an individual and all personal financial information about an individual must be considered confidential. In this example, the following would be the assets:

- Customer information specifically identifying an individual (name, identification number, date of birth).
- Customer financial information (salary, contributions to retirement plans).
- The database where the information is stored.
- The proprietary business rules.
- The process that gathers the data and stores it in a database.

The application allows employees of the organization to modify customer data. It also allows customers to view, but not modify or calculate, their retirement benefits online. Therefore, threats to such assets might include the following:

- Inadvertent, unauthorized modification of data (if customer data was modified or entered by mistake, the program could calculate incorrect benefits).
- Deliberate unauthorized exposure of the data (if a malicious person, either external or internal was able to break into the application).
- Deliberate unauthorized modification of the data (if a malicious individual or process was able to access the database, the data could be destroyed or corrupted).

Likelihood of Vulnerability being Exercised	Criteria Used to Determine Likelihood
High	• Threat source highly motivated and capable. • Controls preventing vulnerability ineffective or non-existent.
Medium	• Threat source highly motivated and capable. • Controls may prevent or at least impede vulnerability. OR • Threat source lacks motivation or capability. • Controls preventing vulnerability ineffective or non-existent.
Low	• Threat source lacks motivation or capability. • Controls prevent or significantly impede vulnerability.

Table 3: *Likelihood Determination* [12]

Impact	Criteria
High	• Costly loss of major tangible assets or resources. • May significantly violate, harm, or impede mission, reputation, or interest. • May result in human death or serious injury.
Medium	• Costly loss of assets or resources. • May violate, harm, or impede mission, reputation, or interest. • May result in human injury.
Low	• Loss of some assets or resources. • Noticeably affect mission, reputation, or interest.

Table 4: *Impact Determination* [12]

Next we need to identify potential vulnerabilities. Upon reviewing a list of the top 10 vulnerabilities from the OWASP site, we see that Structured Query Language (SQL) injection is a possible vulnerability for this application. If validation rules on input fields do not limit the ability to input malicious characters, such as apostrophes, and error-handling routines generate messages to the user that are not edited, then the possibility exists that a malicious user could see the name of the database and potentially some of the field names as well. Armed with this information, a malicious user could potentially alter, destroy, or disclose confidential customer data.

Let us examine the threat-vulnerability pair of a deliberate unauthorized modification of the data by an employee exercising the SQL injection vulnerability. To understand the potential problem, see Figure 2 (see page 24).

In this example, the application's control (code which could prevent the SQL injection from being exercised) is inadequate to protect the customer database (our asset) from the employee (our threat). If the threat were to enter the right character string into the application, the SQL injection vulnerability would reveal information about the customer database directly to the employee (our threat). Therefore risk exists that an employee (threat) could then use that knowledge (vulnerability) to directly alter, disclose, and/or destroy data in the customer database (our asset). But how great is this risk?

Based upon our criteria, the likelihood that this threat-vulnerability pair would be exercised is rated as *medium*. The reason that the likelihood is ranked at medium is that there are no controls in place to prevent the SQL injection from being exercised; however, the possibility that an employee (the only ones who can input

Table 5: *Risk Determination*

Threat Likelihood	Impact		
	Low	***Medium***	***High***
Low	Low Risk	Low Risk	Medium Risk
Medium	Low Risk	Medium Risk	Medium Risk
High	Medium Risk	Medium Risk	High Risk

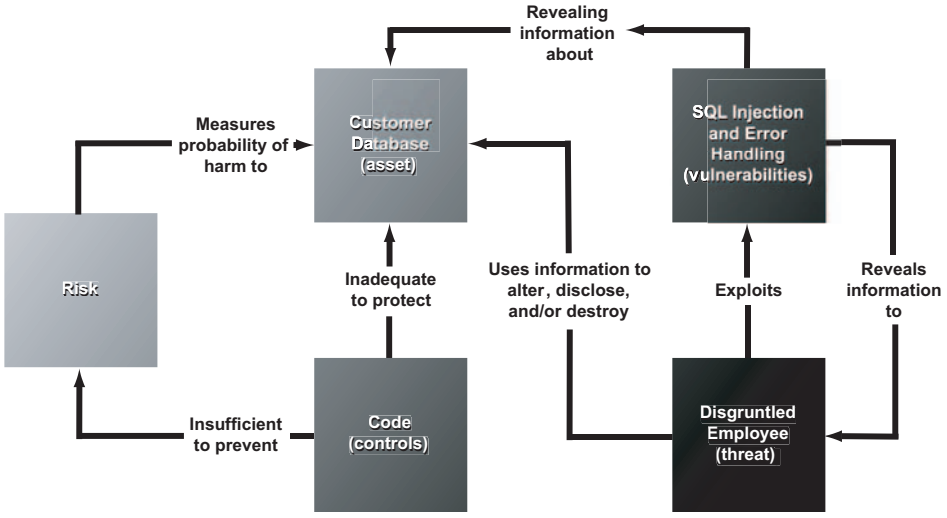


Figure 2: *Example – Risk Exists*

data in the application) is highly motivated and has the knowledge to actively seek out the database and impact it is negligible.

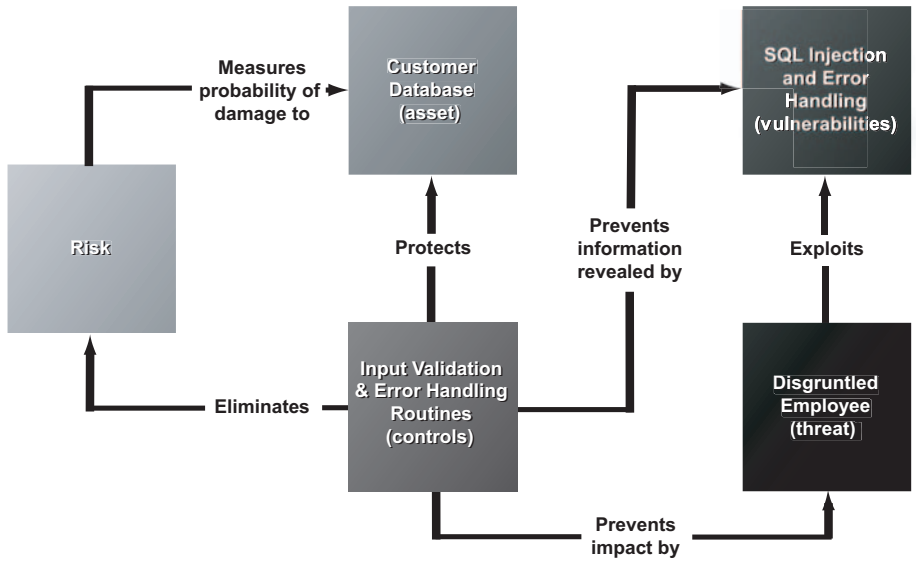
Next, we need to look at the potential impact of the threat-vulnerability pair being exercised. Our threat – the malicious user, through the SQL injection error – knows the name and fields within the database which contain confidential customer information. The malicious user could find customer salaries and social security numbers within this database and place them on the Web, affecting confidentiality. He could alter the information, affecting integrity. Or he could delete all of the information, affecting the availability of the data for others. In this case, we have to assume the worst: The malicious user would reveal the confidential information to others. Depending upon the customer, this could significantly violate the organization's reputation and mission. Therefore, the impact for this threat-vulnerability pair is rated high.

Cross-indexing the threat likelihood with the impact, we see that this is a medium risk. Management can now prioritize this risk alongside all other risks and determine what risk mitigation strategies may be appropriate.

Assume that management has decided to mitigate this risk by requiring that code be developed to prevent a SQL injection attack by requiring validation on all user inputs and only allowing certain error messages to reach end-users (see Figure 3).

In this instance, our control (the code to require proper input validation and error handling) prevents our threat (the employee) from exploiting the SQL injection vulnerability. The employee may attempt to enter the same character string as before, but the code that handles input validation would prevent him from either entering or executing a command using that character string. As a result, no information about the customer database is passed to the employee and he is unable to directly alter, disclose, and/or destroy data in the customer database.

Figure 3: *Example – Risk Mitigated*



Conclusion

Organizations continue to entrust precious assets to software that is developed with the same vulnerabilities over and over again. Ninety percent of software security vulnerabilities are caused by known software defect types, and the top 10 causes account for about 75 percent of all vulnerabilities. How can we reduce these vulnerabilities?

One answer is to apply the practice of assessing security risks within the SDLC. By assessing risks – common threats, vulnerabilities, and risks can be identified. Once the risks are known, then steps can be taken to eliminate or at least mitigate them. Unknown risks can not be eliminated or mitigated. That is why it is important we attempt to detect and analyze risks within the SDLC.◆

References

1. National Institute of Science and Technology. Risk Management Guide for Information Technology Systems. Special Paper 800-30. U.S. Department of Commerce. July 2002 <www.csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf>.
2. Petersen, Gunnar. "Phasing Security Into the SDLC – A Comparison of Approaches." Risk Management. 10 Jan. 2006 <http://1raindrop.typepad.com/1_raindrop/2006/01/phasing_securit.html>.
3. National Cyber Security Partnership. Improving Security Across the Software Development Lifecycle – Task Force Report. 1 Apr. 2004 <www.cyberpartnership.org/SDLCFULL.pdf>.
4. National Institute of Standards and Technology. Database of Threats and Countermeasures. 1999 <http://csrc.nist.gov/nissc/1999/proceeding/papers/o28.pdf>.
5. "@RISK: The Consensus Security Alert." SysAdmin, Audit, Network, Security (SANS) Institute. 5 June 2006 <www.sans.org/>.
6. "CMS Information Systems Threat Identification Resource." Centers for Medicaid and Medicare Services (CMS). 7 May 2002 <http://csrc.nist.gov/fasp/FASPDocs/risk-mgmt/Threat_ID_resource.pdf>.
7. National Cyber Security Partnership. Improving Security Across the Software Development Life Cycle – Task Force Report. 1 Apr. 2004

<www.cyberpartnership.org/SDLCFULL.pdf>.

8. Open Web Application Security Project (OWASP). "Top Ten Most Critical Web Application Security Vulnerabilities." <www.owasp.org/documentation/topten.html>.
9. The MITRE Corporation. "Common Vulnerabilities and Exposures." <http://cve.mitre.org/>.
10. National Institute of Standards and Technology. National Vulnerability Database <http://nvd.nist.gov/>.
11. Lavenhar, Steven, and Gunnar Petersen. "Architectural Risk Assessment." Cigital Inc. 2005 <https://buildsecurityin.us-cert.gov/portal/article/bestpractices/architectural_risk_analysis/architectural_risk_assessment.xml>.
12. National Institute of Standards and Technology (NIST). Risk Management Guide for Information Technology Systems. Special Paper 800-30, July 2002 <www.csrc.nist.gov/publications/nistpubs/800-30/sp80030.pdf>.

About the Author

Douglas A. Ashbaugh, CISSP, is a senior information security analyst for Enterprise Security Services where he provides information security analysis and remediation, policy and procedure development, and security awareness training to various clients. Ashbaugh has a Bachelor of Science in Engineering Operations from Iowa State University. He served eight years in the U.S. Air Force as an acquisition project officer and has worked as a software developer/analyst for the financial services industry.

Enterprise Security Services
1508 JF Kennedy DR STE 201
Bellevue, NE 68005
Phone: (402) 292-8660
Fax: (800) 660-5329
E-mail: dashbaugh@sessolutions.com

Letter to the Editor

Dear CrossTalk Editor,

In the March 2006 issue, Yuri Chernak, in his article *Understanding the Logic of System Testing*, outlined a possible methodological similarity between software testing and mathematical logic. I feel reality of software testing has a close resemblance to the way that experimental science works.

Some mathematical proofs developed in theory of software testing concerning the inexistence of a constructive criterion (i.e. algorithm) to derive the so-called Ideal Test Suite, (the silver bullet of system testing), state that the main goal of testing is restricted to show the presence of failures because testing can never completely establish the correctness of an arbitrary software system.

Skipping all formal aspects, these theoretical results are well summarized: Program testing can be used to show the presence of bugs, but never to show their absence!

Then the methodology of software testing is well rooted in modus tollens (Latin: mode that denies). Modus tollens is the formal name for contrapositive inference and can also be referred to as denying the consequent – it is a valid form of logical argument.

In fact, in experimental sciences, no number of positive outcomes at the level of experimental testing can confirm a theory, but a single counter-instance shows the scientific theory – from which the implication is derived – to be false.

Then, falsification with experimental implementation of modus tollens is just as essential to software testing as it is to scientific theories. Indeed, no number of passed test suites can prove the correctness of a program, but a single failed test suite shows the program to be incorrect.

Effective testing requires an empirical frame of reference rather than a theoretical one: Software reality is more about science than it is about computer science.

Software testing, for mathematical and theoretical reasons, is firmly set in the framework of experimental sciences and we need to see it in this perspective if we want to increase the comprehension of methodology and practice of software testing.

— Francesco Gagliardi
Department of Physical Sciences
University of Naples, Italy
francesco.gagliardi@na.infn.it

Increasing the Likelihood of Success of a Software Assurance Program

Steven F. Mattern
Apogen Technologies, Inc.

As individual systems and collective system-of-systems become more complex, software-intensive, safety-critical and costly, software assurance becomes extremely important. Software assurance helps to reduce the likelihood of failure in terms of safety-related mishaps, system unavailability, loss of mission accomplishment, or security breech of the system or its related assets. Unfortunately, many program managers think they do not have the luxury of sufficient dollars and schedule to adequately address software assurance in the early phases of the acquisition life cycle. One only has to quantify the cost of one system failure or one system loss to comprehend its ultimate worth. Investing the resources in a software assurance program during the design, code, and test phases of a software development program will significantly reduce the likelihood of costly mishaps, failures, or system breeches during system operations and support.

When exposed to the quagmire of buzzwords, definitions, and terminology attached to most government system acquisitions and software development programs, there are many that apply to software assurance. The ones that come immediately to mind are *return-on-investment*, *value added*, *designing it in*, *system integrity*, *safe*, *reliable*, *available*, *secure*, and *risk managed*. While these words may be in the vocabulary of some design teams, there is usually a lack of passionate dedication in implementation that is attached to a lack of sufficient programmatic funding. Unfortunately, software assurance is one of those attributes of system design that is often not an enforced criterion for delivery, installation, or operation of complex systems. While customers and stakeholders are screaming for delivery of the over-budget, behind-schedule items of interest, software assurance is many times reduced to a *nice to have if there is remaining time and money*.

As our systems and system-of-systems become more complex, mission-critical, and safety-critical, it is definitely time for a paradigm shift toward a commitment to software assurance. Adequate funding and passionate commitment are the first steps toward a successful software assurance goal. Closely coupled with the two is a defined and planned process to be implemented to increase the likelihood of software assurance success. Contrary to some development program philosophies, *hope is not an effective methodology* to ensure this success. Success will be based on a defined and documented (stated) set of criterion to be managed within each of the acquisition life cycle phases. In addition, to fully understand the success criterion for software assurance there has to be a complete understanding of system failure [1]. This will be explored further as the elements of software assurance are defined and discussed. A successful software assurance program can then be measured in terms of cost savings, on-time deliveries, and software that is unlikely to contribute to unintended or undesired behavior. These attributes of success are more likely to be realized on a software development program when the output of the software assurance tasks actually minimizes the number of logical or functional flaws in the design architecture.

There are numerous methods or ways to increase the likelihood of a successful software assurance program for software development projects. These methods can be described as the essential elements of a software assurance program (see sidebar). Individually, each element can and will make a contribution to success. Collectively, they will provide a *knockout punch* in ensuring the likelihood of software assurance.

Essential Elements for Software Assurance

- An Adequately Funded Contract
- Program Management Support
- Defined Common Terminology
- Specialty Engineering Support
- Risk Management
- A Defined Set of Processes, Tasks, and Products
- Qualified Practitioners
- Defined and Applicable Metrics

An Adequately Funded Contract

If the Request for Proposal (RFP), the proposal, and resulting contract are not supportive of software assurance methods, the result is clearly *gloom and doom* for the specialty engineer who is actually tasked to perform the work. This is an age-old problem and it definitely needs modern-day rectification. *Business as usual* seldom makes an impact on the engineering disciplines involved with software assurance processes, tasks, and products. Admitting we collectively have a historical and contractual problem is the first step to successful mitigation or control of the issue. So, what can we do differently?

The Government Role

Government agencies need to get serious about ensuring the likelihood of success in this critical area. Modern development programs contain software-intensive, safety-critical, highly reliable, secure, survivable, and operationally effective requirements to go along with system-of-systems integration. The government needs to remain fully committed to ensure that processes are contractually in place within the software development life cycle to produce software that is safe, secure, reliable, and available to perform as intended. Boehm, Kind, and Turner in the *7 Myths about Software Engineering That Impact Defense Acquisitions* [2] state that these processes must be *architected in*. In addition, it must be adequately proposed and funded. Software assurance should be on the government checklist for RFP development. The government should ask for it specifically to be included in the contract proposal, and it should be part of the criteria for proposal evaluation and source selection. The RFP should ask that software assurance be adequately addressed in the software development plan [3] to be delivered as part of the contract proposal.

The Contractor Role

Historically, contractors seldom address software assurance adequately in proposal preparation and conversely it is definitely not sufficiently addressed in the cost volume of the proposal. This is due primarily to program managers' *perception of worth* for software assurance as compared to the more important, documented, and weighted source selection criteria that are contained in an RFP. If the government

specifically requests that software assurance be included in the software development plan and the associated costs for its implementation, the first step to success is complete. Most contractors are willing, if not able, to include software assurance in their development processes if it is adequately requested in the RFP and funded in the contract.

Program Management Support

Tightly coupled with an adequately funded contract is program and project management support. While a perfectly worded and sufficiently funded contract will provide motivation to the program manager, a demonstrated return on investment helps to solidify the motivational factors for supporting software assurance tasks. Program managers must be able to assess the engineering disciplines involved in software assurance activities and obtain a level of confidence that they are *getting their money's worth* with their implementation. This *money's worth* issue is due to the fact that the assurance activities are implemented to prevent *bad* things from occurring (mishaps, security breeches, and mission failures). Therefore, if nothing bad happens, did the program waste a lot of resources combating a very unlikely set of events? Most people can relate well to the discussions several years ago surrounding the Y2K issue and resources expended to combat this very critical issue. But, when the clock struck midnight and no major catastrophe occurred, it set off a series of discussions about whether we had wasted our critical resources on a non-occurring event. On the other hand, most will agree that one single Y2K national catastrophe would have cost more than we spent fighting against it. A good software assurance program will combat the likelihood of software contributing to catastrophic events.

The essential elements of a software assurance activity that will help add to the program manager's confidence to expend critical resources include *risk management*, including the severity and likelihood of failure; *a defined set of processes, tasks and products* that make sense to the design and test teams; *qualified practitioners* with adequate and demonstrated capability; and *defined and applicable metrics* – believable measurements of progress and completeness.

Defined Common Terminology

Historically, specific words have different meanings for individual design teams or stakeholders. The simple term *software assurance* possesses any number of different definitions among commercial contractors, individual design teams, government agencies, or international organizations. Government agencies and their supporting contractors must begin speaking a common language with defined and documented definitions. This is essential for agencies like the Department of Defense (DoD) to make the necessary strides in the development of system-of-systems that will be deployed, operated, and maintained by any or all of the armed services (and some in commercial airspace or similar environments). Software assurance efforts can and will benefit if this occurs.

To demonstrate the problem, a search was accomplished on the Internet using Google. The objective of the search was to find the *best* definition of software assurance available. Looking through the first 100 hits, the following were the only two definitions worthy of consideration:

- **Software Assurance.** A planned program whereby a customer prepays a percentage of their license price for future software releases [4].
- **Software Assurance.** A planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures [5].

However, by knowing exactly where to look, one can find the following definitions from sources other than Google:

- **Software Assurance.** ... the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle, and that the software functions in the intended matter [6].
- **Software Assurance.** Relates to the level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software [7].

In our Google search, the context of the Microsoft Corporation software assurance is prepaying for future application updates, and NASA's definition is moving toward what we would expect it to be (in 1993). But as we move forward to 2006, we are still faced with either conflicting or dissimilar definitions. These example definitions use the same two words, in sometimes separate environments with demonstrated confusion a possible net result.

For the purpose of this article, let us define software assurance as a *planned and defined set of activities that ensures software is complete in design to include safety, security, and reliability*. These activities ensure that the software functions as intended, not performing unintended or undesired functions, and adheres to a predefined set of standards that can be audited and verified.

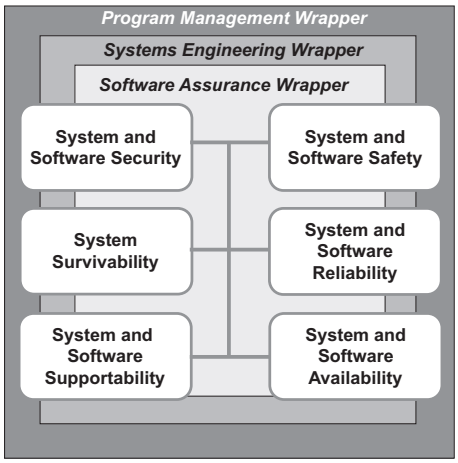


Figure 1: *Specialty Engineering Support*

Specialty Engineering Support

Specialty engineering is a general term used for engineering other than the typical hardware, software, or systems engineers on a design project. It normally includes system safety, reliability, logistics support, human factors, maintainability, security, and survivability. Unfortunately, these specialty engineers on most modern DoD projects do not represent a unified and integrated front that is needed to perform the necessary tasks for the software assurance activity. As described previously, most members of a product team or functional design team find it difficult to even agree on a definition of what software assurance is (or is not).

As depicted in Figure 1, software assurance includes multiple specialty engineering disciplines. Both the *system* and the *software* elements of the discipline are included as we know that system functionality and the hardware and human interface elements of the system are closely coupled with the functionality of the software.

Systems engineering is a key element to the success of the software assurance activity. It is here that the functional analysis is accomplished and that safety, security, and mission-critical functions are identified and managed in the design process. In addition, functional, physical, and logical interfaces are identified and managed as this is where system failure often initiates. The lead systems engineer should be integrating and managing the specialty engineering disciplines and ensuring processes, tasks, and products of each possess value. The lead systems engineer will also ensure

that the specialty engineers analyze the key functions of interest with a focus on both system failure and system success. They will also ensure that there is no duplication of effort and that each discipline is tied into the risk management activities of the program.

Risk Management

Program management is responsible for ensuring that programmatic and technical risk is identified, managed, and mitigated to acceptable levels of risk on a development program. To assist in this task, the *undesired events* of each of the specialty engineering disciplines must be identified and assessed using the risk criticality matrices that are common to risk management methods.

Table 1 [8] provides an example of the *look and feel* of a common criticality matrix. The matrix is used in risk management to categorize and prioritize resources based upon the perception of risk. Risk, in this context, can be in terms of safety, security, availability, reliability, mission risk, and/or supportability. This is all based upon the potential of losing functional or physical attributes of the system, and the command and control the software has over them.

Along with the criticality matrix is the acceptance or management authority over the risk identified. In the example, unresolved high-risk issues could only be accepted at the Program Executive Officer (PEO) level whereas project managers would be able to accept unresolved low-risk issues. The example matrix and the identification of the corresponding acceptance authority provided here are examples only. Individual programs must specifically define and tailor their own matrices.

A Defined Set of Processes, Tasks, and Products

The most important element of software assurance resides in the defined processes, tasks, and products that are produced and implemented by both the software design team and the specialty engineers. Without a defined and approved process, the entire software assurance effort reverts to an ad-hoc effort at best.

The specific disciplines within specialty engineering have one thing in common: they can all identify the *undesired events* that they do not want to occur or experience. In terms of safety, mishaps are the undesired event; in terms of reliability, it is an inoperable system; in terms of security, it is a functional or physical breech leading to system compromise. Regardless of the discipline and the undesired event, there are common tools and techniques to categorize the severity and likelihood of occurrence, identify failure modes and causes, and identify specific system (and subsystem) related requirements to eliminate, mitigate, or control these events or conditions to acceptable levels of risk. Because of this commonality and to simplify the example, only safety will be used within the context of the following discussion to illustrate the process. There are two basic processes to develop safer software: a software safety assurance process and a software safety hazard analysis process.

Software Safety Assurance Process

The Federal Aviation Administration (FAA) has been using a software safety assurance approach to develop safer software for years. This process is predicated on documented software level definitions that define the requirements, design, code, and test criteria to the software development team (refer to the top half of Figure 2). According to the FAA, *the software level is based upon the contribution of software to potential failure conditions as determined by the system safety assessment process* [9]. For the FAA, it is the utilization of RTCA DO-178B that provides the specific criteria for software functionality Levels A through E in descending order of consequence severity (Note: RTCA organized in 1935 as the Radio Technical Commission for Aeronautics, but it is now known as RTCA Inc.). Level A functionality possesses catastrophic severity consequences if it were to be lost, degraded, or if it functioned out of time or out of sequence. Each level possesses a lesser consequence of failure. Level B functionality possessed *critical* consequences whereas Level E functionality possessed no safety impact should it fail. Once the safety team defines the software level definition of a specific function, the software development teams implement the design, code, and test criteria required for FAA certification. This process has numerous benefits, including the following:

- The identification and categorization of functionality based upon safety consequences.
- Increased levels of development and test rigor for high-consequence functionality.
- The functional and physical partitioning of high-consequence functionality to reduce the likelihood of non-critical functions contributing to catastrophic or critical failure.
- The prioritization of critical resources (dollars, schedule, manpower) based upon sound risk management principles.
- *Safer* software and thus safer systems.

Software Safety Hazard Analysis Process

The DoD has relied on a hazard analysis process to develop safer software (refer to bottom half of Figure 2). This approach relied on the identification of system-level mishaps and their corresponding hazards. Mishaps and hazards are then categorized in terms of severity and likelihood of occurrence. By analyzing *snapshots* of a design as it matures, specific hardware, software, and human error causal factors are identified. Once these causal factors are identified in the context of the hazard initiation and failure pathway to potential mishap, then specific safety-related software requirements can be identified to mitigate or control the hazard to acceptable levels of safety risk. These safety-

Table 1: *Example of a Criticality Matrix* [8]

Risk Management – Criticality Matrix				
Probability	**Severity**			
	Catastrophic	Critical	Marginal	Negligible
Frequent	1	3	8	15
Probable	2	5	11	16
Occasional	4	7	12	18
Remote	6	10	14	19
Improbable	9	13	17	20

1-5	High Risk	Accepted — PEO Level
6-12	Medium Risk	Accepted — Program Manager Level
13-18	Low Risk	Accepted — Project Manager Level
19-20	Very Low Risk	Accepted — Systems Engineer Level

related requirements can be in the form of specific design architecture changes, or the addition of fault detection, fault tolerance, or fault recovery. These requirements are traced through the design implementation, the coded product, and then to the test and verification efforts. Benefits of this approach include the following:

- Safety-critical functions can be graphically modeled for ease of in-depth safety analysis [10].
- The causes of failure can be identified and mitigated at their specific initiation points.
- Failure of software functionality is analyzed in context with its hardware and human interfaces.
- Fault detection, isolation, annunciation, tolerance, and recovery can be more precise in its design implementation.
- *Safer* software, thus a safer system.

Integrating the Two Processes

The system safety community is beginning to realize that each of the two processes does indeed result in safer software. But, to obtain the *safest* software possible, modern-day developments need to integrate both the software safety assurance and the software safety hazard analysis processes into the software development and test activities [11].

Safety is used in this example of processes because the software safety community may have more mature methods, tools, and techniques to address software assurance. Because the software safety community processes mature tools and techniques, each specialty engineering discipline should closely evaluate what the safety community is using as it would either directly apply or could be modified slightly to yield exceptional results.

The Tasks and the Products

The defined processes for software assurance represents the *what* that needs to be accomplished. The specific *how to* tasks to fulfill the process are defined and implemented by individual teams. There are many ways to accomplish these tasks as long as the process is being followed. Specific tools such as Fault Tree Analysis and Failure Modes and Effects Analysis are common to the industry and provide the means to completely understand the context of failure. In addition, it is important that the tasks accomplished produce the engineering evidence and audit trail *products* for either system certification or customer acceptance.

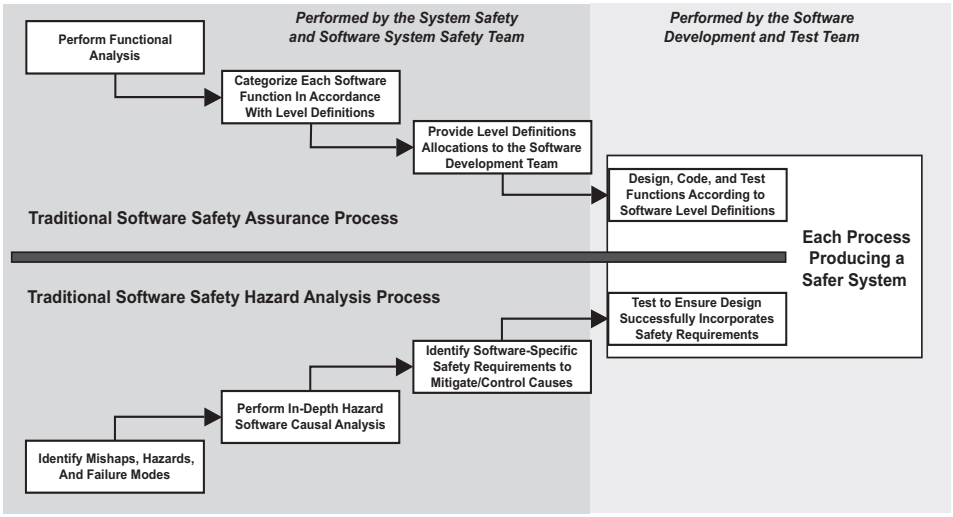


Figure 2: *System Safety and Software Safety Assurance Processes*

Qualified Practitioners

Specialty engineering is not a profession where *body count* is the most important factor. One good, trained, qualified performer is much better than three untrained, struggling practitioners. Contractors have a bad habit of putting engineers with dwindling contractual coverage into positions with specialty engineering just to keep them employed. This is far from optimal and should not be accepted by the stakeholder of the contract.

Also, a lesson learned (and re-learned over and over again) is associated with the original designer being tasked to do the safety or reliability analysis due to the lack of a qualified person being on staff. To put it bluntly, asking the designer to perform failure analysis on their own design is like asking a brand new mother to identify and document how ugly her new baby is. Just as new mothers only see the beauty of their new child, design engineers only see the natural beauty of their design. Asking them to see or identify failure modes and conditions (ugliness) of their design will historically not yield the true results required for a high-fidelity failure analysis.

The bottom line here is that specialty engineers (system safety, reliability, vulnerability, security, etc.) are trained in the art and science of systematically breaking the system down to identify the failure conditions and their contributing failure pathways and initiators. They are trained to categorize and prioritize risk based upon severity and likelihood of occurrence in order to facilitate wise decision making from management and design engineering in risk mitigation and control. These experts are the individuals that should be doing the work.

Defined and Applicable Metrics

In order to confirm or obtain confidence in a defined software assurance process, there must be applicable metrics that record the score of *how well we are doing*. Applicable metrics provide the technical and managerial decision makers with certainty that the resources expended (upfront in the development process) are actually providing the necessary value to the development effort. In addition, these provide the ultimate stakeholder or customer with the confidence that the system is meeting its assurance requirements or objectives.

Here again, it is important to have qualified individuals for any given project who are adept at establishing credible and verifiable metrics. To support this premise, one must consider that many customers desire or require quantified (or quantifiable) metrics that are supported by engineering evidences and artifacts. It is extremely important to have high-fidelity, quantified results that can be supported and verified by a repeatable process.

On a recent project, my company used a specific tool to provide a quantifiable set of metrics to the customer. This tool is modifiable whereby the metric outputs are based upon the inputs of specific assessment criterion. By evaluating the number and types of findings, the level of risk mitigations and controls, and the overall fault tolerance of the system, a specific output score is generated. While this effort was directed at one phase of the acquisition life cycle, tools like this can be modifiable and used in each of the software life cycle phases.

Regardless of the tools or metrics used, they should include ways to measure each functional discipline of the software

assurance activity. That is, safety, security, reliability, etc. should all be assessed, measured, and scored for decision-making and auditing purposes.

Summary

Software assurance is a maturing discipline that is vital for complex software development projects that possess safety, security, reliability, and mission critical attributes. Although the essential elements for a software assurance program presented here are described in a basic abstract format, each element should be further defined, expanded, and refined for individual DoD software-development projects. Each element presented is important to the success formula. However, if one element of this formula is absent, do not let that hinder the inclusion of the remaining elements. By implementing these elements, program, project, and technical managers can increase the likelihood of having a defendable and high fidelity software assurance program.◆

References

1. Raheja, Dev G. Assurance Technologies – Principles and Practices. McGraw-Hill, 1991.
2. Boehm, B., Peter Kind, and Richard Turner. "Risky Business; 7 Myths about Software Engineering That Impact Defense Acquisitions." Project Manager (May-June 2002).
3. Rosenberg, Linda H. "Lessons Learned in Software Quality Assurance." Software Tech News 6.2 (Dec. 2002).
4. Semilof, Margie. "Microsoft Licensing: A Special Report: Licensing 6.0 and Software Assurance: Who Should Renew?" 16 Sept. 2003 <http://searchwinit.techtarget.com/originalContent/0,289142,sid1_gci927989,00.html>.
5. National Aeronautics and Space Administration. "Software Assurance Standard: NASA-STD-2201-93." 1992.
6. Committee on National Information Security Systems. "National Information Assurance Glossary." Instruction No. 4009, 2006 <www.cnss.gov/Assets/pdf/cussi_4009.pdf>.
7. Department of Defense (DoD). "DoD Software Assurance Initiative." 13 Sept. 2005 <https://acc.dau.mil/CommunityBrowser.aspk?id=25749>.
8. Mattern, Steven F. "Introduction to Risk Management." System Safety Management Course. University of Washington, Seattle, WA. Mar. 2006.
9. RTCA/DO-178B. "Software Considerations in Airborne Systems and Equipment Certification. Requirements and Technical Concepts for Aviation." 1 Dec., 1992.
10. Mattern, S, E. Elcock, and E. Larsen. "IMPACT – A New Tool for the Software Safety Engineering Toolbox, Integrated Message and Process Analysis Control Technique." Proc. 20th International System Safety Society Conference Denver, CO, 2002.
11. Mattern, Steven F. "Comparing Software Safety Engineering with Software Integrity Methods and Techniques The Implications to Future." Department of Defense Acquisitions, 22nd International System Safety Society Proceedings. Providence, RI, 2004.

About the Author

Steven F. Mattern is vice president with Apogen Technologies, in McLean, VA. He manages the Software and Systems Analysis Division that specializes in software development and software assurance technologies. Engineers in his division have been performing software assurance-related tasks for more than 12 years for both government and commercial clients. Mattern is a Fellow member of the International System Safety Society and holds the position of Director of Education and Professional Development for the Society. He is the integrating author of the *Tri-Services Software System Safety Handbook*, and currently teaches the System Safety Management and Software Safety Engineering Courses at the University of Washington. Mattern has a Bachelor of Science degree in Industrial/Electronic Technology from the University of Wyoming and a Master of Arts in Computer Resource Management from Webster University.

Apogen Technologies, Inc.
1308 Bellevue BLVD N
Bellevue, NE 68005
Phone: (402) 502-3657
E-mail: steve.mattern@apogen.com

Ready, Fire, Aim!

First things first: In my July 2006 BACKTALK article, "e-Dorado: The Lost Centric City of Information," the Office of the Secretary of Defense (OSD) net-centricity quote and reference were incorrect. The word *censors* should have been *sensors* and the reference should have been the OSD Net-Centric Checklist [1]. Ironically, only one astute reader caught the faux pas. In the wake of the Patriot Act and the National Security Agency's warrantless surveillance controversy, can I presume most of you thought censorship was part of net-centric warfare?

With quality on my mind, let's move on to this issue's theme – Software Assurance. The DoD relates software assurance *to the level of confidence that software functions as intended and is free of vulnerabilities* ... [2]. NASA adds that software assurance is *a planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures* [3]. The essence of software assurance is predictability, trustworthiness, and conformance. Trustworthiness and process conformance receive plenty of press, so I will leave them to the security and process enthusiasts and focus on predictability.

With regard to predictability, the DoD wants software that is accurate (functions as intended) and precise (with confidence). I know, many readers are saying, "Gary, accuracy and precision are synonymous." Well, maybe in English 101, but not in science, engineering, and, as it turns out, marksmanship.

Several years ago, my son, Matthew, decided to obtain his rifle merit badge. After a safety lecture, we proceeded to the gun range where shooters sit side-by-side, taking 10 shots at individual targets similar to the target in Figure 1. Matthew was somewhat timid, having never handled a gun, so I agreed to shoot with him to ease his doubts.

We methodically fired 10 rounds, waited for the range to clear, and then marched to our targets to check our results. Concerned about my son's tally I watched as his shoulders slumped and his head dropped. This was not a good sign. I looked over his shoulder. His target looked exactly like Figure 1 – clean, no bullet holes, not even one! While considering how to handle the situation, I glanced at my target, which looked like Figure 2, and there was my answer.

Instead of 10 bullet holes, my target had 20. Matthew shot at my target and not his own. I shared my discovery with Matthew and his shoulders went back, his head snapped up, and later that day he earned the merit badge.

Although I could not differentiate my bullet holes from my son's, I did find the target pattern intriguing. Two distinct patterns appeared on the target. Ten holes were grouped around the center bull's-eye and slightly high. The other 10 holes are tightly grouped below and left of the center bull's-eye. In subsequent rounds, we found Matthew's targets to be tightly grouped and below center-left. I deduced that Matthew's shots were precise (his shots produced similar results – tightly grouped holes), but not accurate (his shots were below and left of the center bull's-eye) and my shots were more accurate (around the center bull's-eye) but less precise (not as tightly grouped).

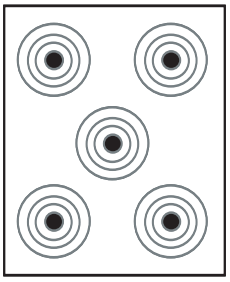


Figure 1: *Individual Target*

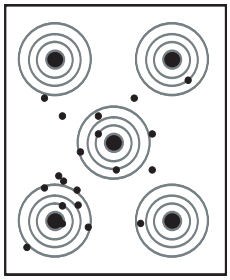


Figure 2: *Twenty Bullet Holes*

Figure 3 is a statistical view of accuracy versus precision. In software development terms, the "Actual Target" in Figure 3 represents the customer's bona fide requirements. The statistical depiction illustrates that you can be precise but not accurate, accurate but not precise or both. The DoD wants both – emulating user bona fide requirements (accuracy) with little variance over time (precision).

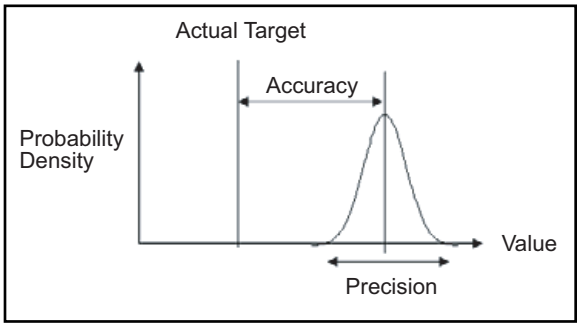


Figure 3: *Accuracy Versus Precision*

Accuracy requires that developers discern true reference values (requirements). However, the capricious nature of software requirements makes that discernment difficult, requiring focus, patience, and clear communication.

That reminds me of the story of the project manager who was late for a design review. Driving to the review, he tried to save time by slowing down and rolling through stop signs. A police officer stopped the project manager and cited him for failure to stop at a stop sign. Being a project manager, he naturally felt he was right and tried to convince the officer that he did slow down at each light. The officer replied, "But you did not stop." The project manager continued his plea, explaining that he slowed down and looked both ways. The officer replied, "But you did not stop." He explained that he slowed down and no one was hurt. The officer replied, "But you did not stop."

Rather than cutting his losses, the project manager, in typical project manager style, pressed the officer to the limit to avoid the ticket. Finally frustrated, the officer pulled the project manager from the car and started beating him with his baton. The project manager started to scream and yelled to the officer, "…stop, please stop!" To which the officer replied, "Do you want me to stop or just slow down?"

What's my point? Aim at the right target then worry about precision, beat your project manager occasionally, and know when to stop.

— Gary A. Petersen
Shim Enterprises, Inc.
gary.petersen@shiminc.com

References

1. Office of the Assistant Secretary of Defense for Networks and Information Integration/Department of Defense Chief Information Officer. "Net-Centric Checklist 2.1.3." 12 May, 2004.
2. Department of Defense Software Assurance Initiative. 13 Sept. 2005.
3. National Aeronautics and Space Administration. "Software Assurance Standard." NASA-STD-2201-93. 10 Nov. 2002.

CrossTalk is co-sponsored by the following organizations:

76th Software Maintenance Group

309th Software Maintenance Group

402d Software Maintenance Group

Homeland Security

NAVAIR

CrossTalk/517 SMXS/MDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737